

3. The ColecoVision Programmers Manual - Detail Sections

TABLE OF CONTENTS

	Page
III. GRAPHICS GENERATION SOFTWARE.....	3-1
3.1. Chip Driver Level.....	3-1
3.1.1 READ_VRAM.....	3-4
3.1.2 WRITE_VRAM.....	3-6
3.1.3 READ_REGISTER.....	3-8
3.1.4 WRITE_REGISTER.....	3-11
3.1.5 FILL_VRAM.....	3-13
3.1.6 MODE_1.....	3-15
3.2 Table Level.....	3-17
3.2.1 Table Managers.....	3-21
3.2.1.1 INIT_TABLE.....	3-22
3.2.1.2 GET_VRAM.....	3-25
3.2.1.3 PUT_VRAM.....	3-29
3.2.2 Table-Oriented Graphics Routines.....	3-32
3.2.2.1 REFLECT_VERTICAL.....	3-35
3.2.2.2 REFLECT_HORIZONTAL.....	3-40
3.2.2.3 ROTATE_90.....	3-45
3.2.2.4 ENLARGE.....	3-51

TABLE OF CONTENTS

	Page
3.2.3 Sprite Reordering Software.....	3-57
3.2.3.1 INIT_SPR_ORDER.....	3-61
3.2.3.2 WR_SPR_NM_TBL.....	3-63
3-3 Object Level.....	3-65
3.3.1 Object Types.....	3-65
3.3.1.1 Semi-Mobile.....	3-66
3.3.1.2 Mobile.....	3-66
3.3.2.3 Sprite.....	3-66
3.3.1.4 Complex.....	3-67
3.3.2 Object Data Structure.....	3-67
3.3.2.1 Graphics Data Area.....	3-67
3.3.2.2 Status Area.....	3-68
3.3.2.3 OLD_SCREEN.....	3-68
3.3.3 ACTIVATE.....	3-69
3.3.4 PUTOBJ.....	3-71
IV. INTERRUPT HANDLING AND WRITE DEFERRAL.....	4-1
4.1 INIT_WRITER.....	4-3
4.2 WRITER.....	4-5

TABLE OF CONTENTS

	Page
V. TIMING SOFTWARE.....	5-1
5.1 Non-Repeating Timer.....	5-2
5.2 Repeating Timer.....	5-2
5.3 TIMER_TABLE.....	5-2
5.3.1 Mode_Byte.....	5-3
5.3.2 Value_Word.....	5-3
5.3.3 TIMER_DATA_BLOCK.....	5-4
5.4 INIT_TIMER.....	5-5
5.5 TIME-MGR.....	5-7
5.6 REQUEST_SIGNAL.....	5-9
5.7 TEST_SIGNAL.....	5-12
5.8 FREE_SIGNAL.....	5-14
VI. CONTROLLER SOFTWARE.....	6-1
6.1 Controller Data Area.....	6-2
6.2 POLLER.....	6-6
6.3 DECODER.....	6-8
6.4 CONT_SCAN.....	6-10
6.5 UPDATE_SPINNER.....	6-11

TABLE OF CONTENTS

	Page
VII. SOUND GENERATION SOFTWARE.....	7-1
7.1 LST_OF_SND_ADDRS and PTR_TO_LST_OF_SND_ADDRS...	7-2
7.2 SOUND_INIT.....	7-2
7.3 PLAY_IT.....	7-4
7.4 SOUND_MAN.....	7-6
7.5 PLAY_SONGS.....	7-8
7.6 Application.....	7-9
VIII. BOOT_UP SOFTWARE.....	8-1
8.1 Power-Up Procedure.....	8-1
8.2 Title Screen.....	8-3
8.3 Cartridge Present Identifier.....	8-3
IX. MISCELLANEOUS UTILITIES.....	9-1
9.1 ADD816.....	9-1
9.2 DECLSN.....	9-3
9.3 DECMSN.....	9-4
9.4 MSNTOLSN.....	9-5
9.5 RAND_GEN.....	9-6
9.6 LOAD_ASCII.....	9-7

TABLE OF CONTENTS

	Page
X. DEFINED REFERENCE LOCATIONS.....	10-1
10.1 OS ROM.....	10-1
10.1.1 Europe/America Byte.....	10-3
10.1.2 Restart Vectors.....	10-5
10.2.3 Graphics Tables.....	10-6
10.2 CART_ROM.....	10-6
10.3 CRAM Areas.....	10-8

APPENDICES

A. Bibliography
B. Graphics Documentation
C. Sound Documentation
D. ColecoVision Bulletins
E. Jump Table
F. OS_SYMBOLS
G. Timing Software Data Structure
H. OS Subroutine Library
I. ASCII Generator Set

SECTION III

GRAPHICS GENERATION SOFTWARE

The graphics generation process is structured on three levels of software. A typical application will use routines from all three levels. These are the chip driver level, the table level and the object level. Figure 3-1 shows the program flow, software structure and its relationship with the outside world.

3.1 Chip Driver Level

The graphics hardware consists of the VDP and 16K VRAM. The VDP has eight write-only control registers and one read-only status register. The chip driver level software interfaces with the VDP registers and VRAM through the VDP. For detailed configuration of the registers, refer to the TMS 9928A VDP Data Manual.

The chip driver level software consists of six subroutines:

1 READ_VRAM, WRITE_VRAM, READ_REGISTER, WRITE_REGISTER,
2 FILL_VRAM and MODE_1. The first five routines allow
3 programs to access the VDP registers and transfer
4 information to and from VRAM blocks. The sixth routine,
5 MODE_1, initializes the VDP into a standard
6 configuration.

7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

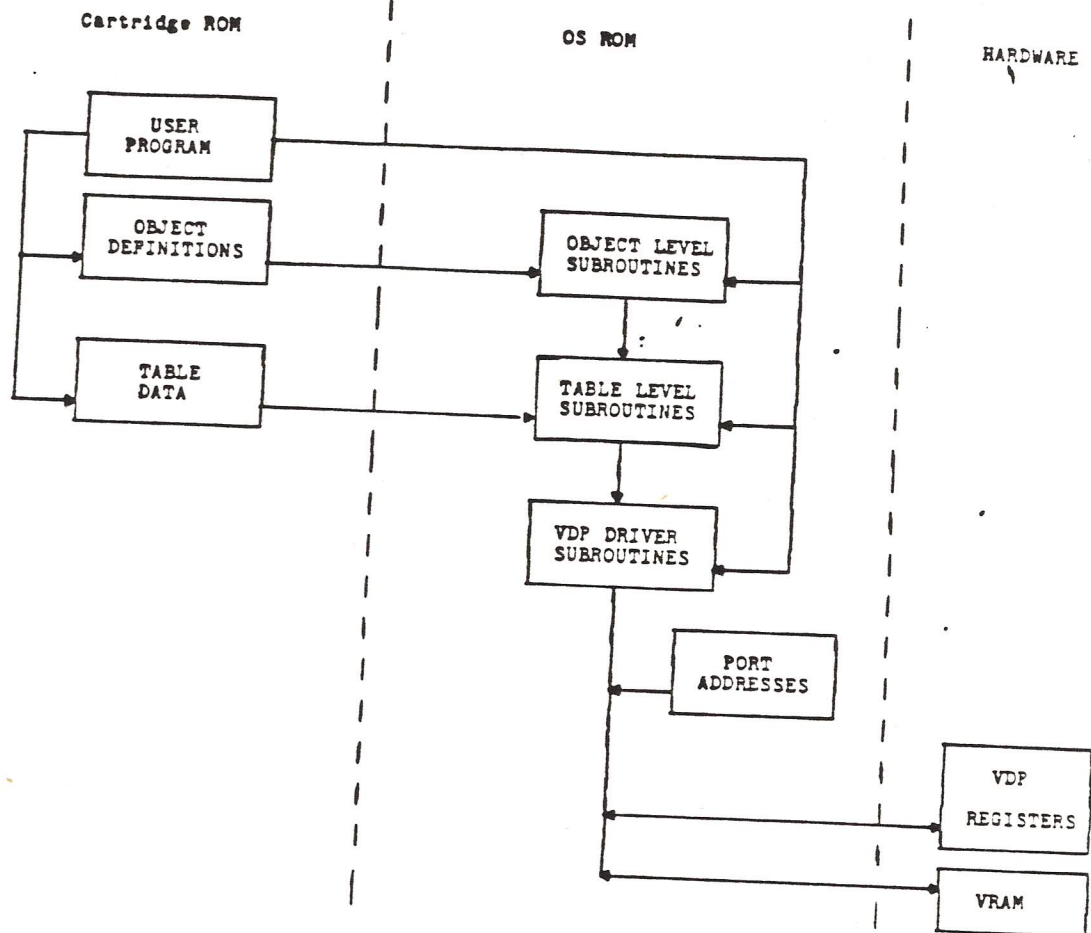


Figure 3-1
OS Graphics Software/VDP Interface

1
2 3.1.1. READ_VRAM

3
4 Calling Sequence:

5
6 LD HL, BUFFER

7 LD DE, SRCE

8 LD BC, COUNT

9 CALL READ_VRAM

10
11 Description:

12
13 READ_VRAM reads COUNT bytes from VRAM starting at SRCE
14 and puts them in BUFFER.

15
16 Parameters:

17
18 BUFFER

This is the starting address of a
CRAM buffer which is to receive
the data read from VRAM.

19
20
21
22 SRCE

VRAM starting address to be read
from.

23
24

25

26

1

COUNT

Number of bytes to be read from

2

VRAM.

3

4

Side Effects:

5

6

- Destroys AF, BC, DE and HL.

7

- Cancels any previously initiated VDP operations.

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

1

2

3.1.2 WRITE_VRAM

3

4

Calling Sequence:

5

6

LD HL, BUFFER

7

LD DE, DEST

8

LD BC, COUNT

9

CALL WRITE_VRAM

10

11

Description:

12

13

WRITE_VRAM takes COUNT bytes from BUFFER and sends them through the VDP to VRAM. The starting address in VRAM for the write operation is given as DEST.

14

15

16

17

Parameters:

18

19

BUFFER

This is the starting address of a buffer where data to be sent to the VDP is located.

20

21

22

23

24

25

26

1		
2	DEST	This is the VRAM address where the
3		data is to be sent.
4		
5	COUNT	This is the number of bytes that
6		are to be transferred to VRAM.
7		Count should be either less than
8		256 (100H) or even multiples of
9		256. (Ref. ColecoVision Bulletin
10		No. 0002).
11	Side Effects:	
12		
13		- Destroys AF, BC, DE and HL.
14		- Cancels any previously initiated VDP operations.
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		

1

2

3.1.3 READ_REGISTER

3

4

Calling Sequence:

5

6

CALL READ_REGISTER

7

8

Description:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

Returns the contents of the VDP status register which has the following form (see VDP manual for further details):

4.1.7 EOS Entry Points

```

ADD816          EQU 0FD4DH :P
BLK_STRT_PTR    EQU 0FDDCH :D
BLOCKS_REQ      EQU 0FE0CH :D
BUF_END         EQU 0FE0AH :D
BUF_START       EQU 0FE08H :D
BYTES_REQ       EQU 0FE02H :D
BYTES_TO_GO     EQU 0FE04H :D
CALC_OFFSET     EQU 0FD32H :P
CLEAR_RAM_SIZE  EQU 00147H :A
CLEAR_RAM_START EQU 0FD60H :D
COLORTABLE      EQU 0FD6CH :D
CONTROLLER_0_PD EQU 0FC2BH :A
CONTROLLER_1_PD EQU 0FC2CH :A
CURRENT_DEV     EQU 0FD06H :D
CURRENT_PCB     EQU 0FD70H :D
CURSOR         EQU 0FEA5H :D
CUR_BANK       EQU 0FD6EH :D
DCB_IMAGE      EQU 0FD8BH :D
DECLSN         EQU 0FD44H :P
DECMSN         EQU 0FD47H :P
DEFAULT_BT_DEV  EQU 0FD6FH :D
DEVICE_ID       EQU 0FD72H :D
DIR_BLOCK_NO    EQU 0FDD9H :D
EFFECT_OVER     EQU 0FD5CH :P
EOS_DAY         EQU 0FDE2H :D
EOS_MONTH       EQU 0FDE1H :D
EOS_STACK       EQU 0FE58H :D
EOS_YEAR        EQU 0FDE0H :D
PCB_BUFFER      EQU 0FDBAH :D
FCB_DATA_ADDR   EQU 0FDFFH :D
FCB_HEAD_ADDR   EQU 0FDFDH :D
FILENAME_CMPS   EQU 0FDD8H :D
FILE_COUNT      EQU 0FDD4H :D
FILE_NAME_ADDR  EQU 0FD73H :D
FILE_NUMBR      EQU 0FDD7H :D
FILL_VRAM       EQU 0FD26H :P
FMGR_DIR_ENT    EQU 0FDE3H :D
FNUM           EQU 0FE01H :D
FOUND_AVAIL_ENT EQU 0FDD8H :D
GET_VRAM        EQU 0FD2FH :P
INIT_TABLE      EQU 0FD29H :P
INT_VCTR_TBL    EQU 0FBFFH :A
KEYBOARD_BUFFER EQU 0FD75H :D
LINEBUFFER_     EQU 0FE7EH :D
LOAD_ASCII      EQU 0FD38H :P
MEM_CNFG00      EQU 0FC17H :A
MEM_CNFG01      EQU 0FC18H :A
MEM_CNFG02      EQU 0FC19H :A
MEM_CNFG03      EQU 0FC1AH :A
MEM_CNFG04      EQU 0FC1BH :A
MEM_CNFG05      EQU 0FC1CH :A
MEM_CNFG06      EQU 0FC1DH :A
MEM_CNFG07      EQU 0FC1EH :A
MEM_CNFG08      EQU 0FC1FH :A
MEM_CNFG09      EQU 0FC20H :A
MEM_CNFG0A      EQU 0FC21H :A
MEM_CNFG0B      EQU 0FC22H :A
MEM_CNFG0C      EQU 0FC23H :A
MEM_CNFG0D      EQU 0FC24H :A
MEM_CNFG0E      EQU 0FC25H :A

```

```

MEM_CNFG0F      EQU 0FC26H :A
MEM_SWITCH_PORT EQU 0FC27H :A
MOD_FILE_COUNT  EQU 0FDD5H :D
MSNTOLSN       EQU 0FD4AH :P
NET_RESET_PORT  EQU 0FC28H :A
NEW_HOLE_SIZE   EQU 0FE1AH :D
NEW_HOLE_START  EQU 0FE16H :D
NUM_COLUMNS     EQU 0FEA0H :D
NUM_LINES       EQU 0FE9FH :D
JLDCHAR        EQU 0FE79H :D
PATRNGENTBL     EQU 0FD6AH :D
PATRNNAMETBL    EQU 0FD68H :D
PCB            EQU 0FEC0H :A
PERSONAL_DEBOUN EQU 0FE5AH :D
PLAY_IT        EQU 0FD56H :P
POLLER         EQU 0FD3EH :P
PORT_COLLECTION EQU 0FD11H :P
PORT_TABLE      EQU 0FC27H :A
PRINT_BUFFER    EQU 0FD76H :D
PTRN_NAME_TBL   EQU 0FEA3H :D
PTR_TO_LST_OF_S EQU 0FE6EH :D
PTR_TO_S_ON_0   EQU 0FE70H :D
PTR_TO_S_ON_1   EQU 0FE72H :D
PTR_TO_S_ON_2   EQU 0FE74H :D
PTR_TO_S_ON_3   EQU 0FE76H :D
PUT_ASCII       EQU 0FD17H :P
PUT_VRAM        EQU 0FD2CH :P
PX_TO_PTRN_POS  EQU 0FD35H :P
QUERY_BUFFER    EQU 0FDA0H :D
READ_REGISTER   EQU 0FD23H :P
READ_VRAM       EQU 0FD13H :P
RETRY_COUNT     EQU 0FDD6H :D
REV_NUM         EQU 0FD60H :D
SAVE_CTRL       EQU 0FE78H :D
SECTORS_TO_INIT EQU 0FD86H :D
SECTOR_NO       EQU 0FD87H :D
SCNDPRT         EQU 0FC2FH :A
SOUNDS          EQU 0FD59H :P
SOUND_INIT      EQU 0FD50H :P
SPIN_SWO_CT     EQU 0FE58H :D
SPIN_SWI_CT     EQU 0FE59H :D
SPRITEATTRTBL  EQU 0FD64H :D
SPRITEGENTBL    EQU 0FD66H :D
START_BLOCK     EQU 0FE12H :D
STROBE_RESET_PD EQU 0FC2EH :A
STROBE_SET_PORT EQU 0FC2DH :A
SWITCH_MEM      EQU 0FD14H :P
SWITCH_TABLE    EQU 0FC17H :A
TEMP_STACK      EQU 0FE6EH :D
TURN_OFF_SOUND  EQU 0FD53H :P
UPDATE_SPINNER  EQU 0FD41H :P
UPPER_LEFT      EQU 0FEA1H :D
USER_BUF        EQU 0FE06H :D
USER_NAME       EQU 0FE10H :D
VDP_CTRL_PORT   EQU 0FC29H :A
VDP_DATA_PORT   EQU 0FC2AH :A
VDP_MODE_WORD   EQU 0FD61H :D
VDP_STATUS_BYTE EQU 0FD63H :D
VECTOR_08H      EQU 0FBFFH :A
VECTOR_10H      EQU 0FC02H :A
VECTOR_18H      EQU 0FC05H :A
VECTOR_20H      EQU 0FC08H :A
VECTOR_28H      EQU 0FC0BH :A

```

Bit 7	Bit 6	Bit 5	Bits 4..0
Interrupt	Fifth Sprite	Coincidence	Fifth Sprite No.

Figure 3-2

VDP Status Register

Side Effects:

This routine has no effect at all in the processor memory or register space. However, a status read has a significant side effect to the VDP.

It acts as an interrupt acknowledge operation, i.e., it clears the interrupt flag and enables further generation of interrupts.

This side effect must be treated with care for two reasons. First of all, as is pointed out in the VDP manual, asynchronous reads may cause the interrupt flag in the status register to be reset before it is detected; this may cause problems in systems that expect to perform synchronization using the interrupt flag.

1 The second reason concerns interrupts which halt the
2 execution of routines while they are accessing VRAM. In
3 order to re-enable interrupts, a service routine must
4 read the status register. However, to prevent the NMI
5 from re-interrupting the service routine, the user
6 should avoid reading the status register until all of
7 its work is done. A defer interrupt routine, DEF_INT,
8 has been developed to assist the user in handling this
9 situation. Refer to ColecoVision Bulletin No. 0010 for
10 additional information.
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

3.1.4 WRITE_REGISTER

Calling Sequence:

```
LD    B, REGISTER
LD    C, VALUE
CALL  WRITE_REGISTER
```

Description:

WRITE_REGISTER takes VALUE and writes it to the VDP register numbered REGISTER.

WRITE_REGISTER also maintains two bytes in CRAM starting at address VDP_MODE_WORD. The first is intended to duplicate the current contents of VDP Register 0, and the second to duplicate Register 1. When writing to a register using WRITE_REGISTER, the appropriate half of VDP_MODE_WORD is updated.

1

2

Parameters:

3

4

REGISTER

This is the VDP register number
(0 - 7) to be written.

5

6

7

VALUE

This is the value to be written to
REGISTER.

8

9

10

Side Effects:

11

12

- Destroys the AF register pair.

13

14

15

16

17

18

19

20

21

22

23

24

25

26

1

2

3

3.1.5 FILL_VRAM

4

5

Calling Sequence:

6

7

LD HL, ADDRESS

8

LD DE, COUNT

9

LD A, VALUE

10

CALL FILL_VRAM

11

12

Description:

13

14

FILL_VRAM writes COUNT copies of VALUE to VRAM starting at ADDRESS.

15

16

17

Parameters:

18

19

ADDRESS

VRAM address to start fill operation.

20

21

22

COUNT

Number of bytes to fill.

23

24

25

26

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

VALUE

8-bit value to fill with.

Side Effects:

- Destroys AF and DE.
- Cancels all previously initiated VRAM operations.

Calls to other OS routines:

- READ_REGISTER (Ref. Sec. 3.1.3)

3.1.6 MODE_1

Calling Sequence:

CALL MODE_1

Description:

MODE_1 sets the VDP to graphics mode 1 and sprite size 0. It also uses the INIT_TABLE routine to define the VRAM table addresses as follows:

- Sprite Generator Table	- 3800H
- Patter Color Table	- 2000H
- Sprite Attribute Table	- 1B00H
- Pattern Name Table	- 1800H
- Pattern Generator Table	- 0000H

When MODE_1 returns, the screen is blanked and the backdrop plane color is set to black.

1

2

Side Effects:

3

4

- Destroys AF, BC and HL.

5

6

Calls to other OS routines:

7

8

- WRITE_REGISTER

9

- INIT_TABLE

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

3.2 Table Level

The VDP requires various table areas within VRAM to operate. These tables are interrelated, each controlling its own aspect of the graphics generation process. The table level software provides routines which will read or write VRAM with respect to these table areas. The routines also provide the capability of reading and writing entire tables entries or sections of these entries up to and including the whole table. This level also has special functions which were found helpful.

The major difference between the table level and the chip driver level is that the applications programmer is no longer required to manipulate VRAM addresses on the table level. Instead, each of the VRAM tables is assigned a number or table code as listed in Table 3-1.

Table Name	Code
Sprite attribute table	0
Sprite generator table	1
Pattern name table	2
Pattern generator table	3
Pattern color table	4

Table 3-1
VRAM Table Code

When an applications program needs to operate on a table, only a table code needs to be passed to the applicable table processing the routine.

Furthermore, in graphics mode 1 and graphics mode 2, which are supported by the OS graphics software, the tables have more or less fixed shapes. The entry numbers and bytes per entry for each of the five tables, as well as their boundaries, is given in Table 3-2.

TABLE CODE	MODE(S)	ENTRIES	BYTES/ ENTRY	HEX EQUIVALENTS
0	1 & 2	32	4	80H
1	1 & 2	256	8	800H
2	1 & 2	768	1	400H
3	1	256	8	800H
3	2	768	8	800H
4	1	32	1	40H
4	2	768	8	2000H

Table 3-2
Table Entries and Boundaries

The table management software takes advantage of this regularity by letting application programs address table entries as integral entities. Let us take, for example, the task of getting the 14th sprite attribute entry from VRAM. In terms of the chip driver software, the task appears as follows:

- Get sprite attribute table address.
- Calculate offset into table (14 * table row length).
- Add offset to address.
- Read one table entry (4 bytes) from VRAM at off set + attribute table address.

1 On the other hand, when using the table level software,
2 the task is now reduced to the following:
3

- 4 - Give offset into table (14).
5 - Give table code.
6 - Give item count (1).
7 - Call GET_VRAM (places the desired bytes at a
8 user-defined area).
9

10
11 In a video program that requires accessing the sprite
12 attribute table frequently (for example, an action-
13 oriented game), the table level method constitutes a
14 significant savings in cartridge code.
15

16 Software in the table level may be further subdivided
17 into three groups of routines as follows:
18

- 19 - Table Managers
20 - Table-oriented Graphics Routines
21 - Sprite Reordering Software
22
23
24
25
26

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

3.2.1 Table Managers

There are three routines in this group: INIT_TABLE, GET_VRAM and PUT_VRAM. As the names imply, they deal with table initialization, getting data from tables and placing data into tables, respectively.

Table initialization is a very simple operation which involves assigning a base address to a table. The base addresses are "saved" for later use by GET_VRAM and PUT_VRAM for address calculations and remain fixed until they are reinitialized. GET_VRAM and PUT_VRAM both take a table code, an entry number, as well as an element count and a buffer address in CRAM as parameters when they perform their respective transfers of information between CRAM and VRAM.

3.2.1.1 INIT_TABLE

Calling Sequence:

```
LD    A, TABLE_CODE
LD    HL, ADDRESS
CALL  INIT_TABLE
```

Description:

INIT_TABLE takes a table code and a VRAM address at which that table is to reside, and initializes the VDP base address register for the given table. It also stores the unconverted form of the address in an array called VRAM_ADDR_TABLE for later use in address arithmetic. This address is stored at VRAM_ADDR_TABLE [TABLE_CODE].

INIT_TABLE makes use of the current graphics mode in determining the actual value written to the base address register in some cases. It determines the graphics mode

1 by looking at the VDP_MODE_WORD. Thus, it is imperative
2 that the graphics mode be set up using WRITE_REGISTER
3 before INIT_TABLE.

4
5 Parameters:

6
7 TABLE_CODE Number of the table to be
8 initialized. TABLE_CODE must be
9 one of the legal table codes
10 defined in Table 3-1.

11
12 ADDRESS Intended VRAM address of table.
13 Each table has its own boundary
14 defined by the table base address
15 in the VDP control register. The
16 user should refer to Table 3-2 for
17 the proper table boundary.

18
19 Side Effects:

20
21 - Destroys AF, BC, HL, IX and IY.
22
23
24
25
26

1

Calls to other OS routines:

2

3

- REG_WRITE

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

3.2.1.2 GET_VRAM

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, START_INDEX
LD    HL, DATA
LD    IX, COUNT
CALL  GET_VRAM
```

Description:

GET_VRAM reads into the CRAM buffer DATA, COUNT entries from the table specified by TABLE_CODE, which starts at the table entry number START_INDEX.

GET_VRAM uses the VDP_MODE_WORD and VRAM_ADDR_TABLE to calculate VRAM addresses and byte counts. It is imperative, before calling GET_VRAM, that the graphics mode be initialized using WRITE_REGISTER, and that the table being accessed be initialized using INIT_TABLE.

Parameters:

TABLE_CODE

VRAM table code (Table 3-1) to be read.

START_INDEX

START_INDEX is a two-byte number that indicates the starting entry of the table.

The range of START_INDEX is table dependent. However, no boundary checking is done; therefore, if an index is given that is outside the range of the table, but still a legal VRAM address, the specified number of "entries" will be extracted from that location in VRAM.

Both the pattern generator and the color tables in graphics mode 2 are 768 entries long and they are

segmented into three sections corresponding to the three sections of the display. When addressing these tables, the high order byte (D) of the two-byte START_INDEX value is a "segment specifier" ($0 \leq D \leq 2$), while the low order byte (E) specifies the index of the entry in that segment.

In the case of the sprite generator table, please note that COUNT refers to 8-byte shape for entries whether one is using size 0 or size 1 sprites.

DATA

Starting address of a CRAM data buffer to receive data from VRAM.

COUNT

Number of entries to be read from the VRAM table.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

The restrictions on COUNT are again table dependent. In other words, it should always be the case that $\text{START\_INDEX} + \text{COUNT} \leq \text{Table Size}$.

Side Effects:

- Destroys AF, BC, DE, HL, IX and IY.
- This routine uses the local storage area SAVED_COUNT and is therefore not re-entrant.

Calls to other OS routines:

- READ_VRAM

3.2.1.3 PUT_VRAM

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, START_INDEX
LD    HL, DATA
LD    IY, COUNT
CALL  PUT_VRAM
```

Description:

PUT_VRAM writes from the buffer DATA, COUNT entries to the table specified by TABLE_CODE, which starts at the table entry number START_INDEX.

PUT_VRAM uses the VDP_MODE_WORD and VRAM_ADDR_TABLE to calculate VRAM address and byte counts. It is imperative that the graphics mode be set up using WRITE_REGISTER and the table being accessed be initialized using INIT_TABLE before PUT_VRAM is called.

The table level of graphics software contains a sprite reordering feature where the major effect is in the operation of PUT_VRAM. When the MUX_SPRITES flag is set to TRUE (1), PUT_VRAM writes sprite entries to a CRAM copy of the sprite attribute table instead of writing them to VRAM. It locates this table through a pointer in low cartridge ROM called LOCAL_SPR_TBL. The sprite entries will then be re-ordered before being written to VRAM.

Parameters:

TABLE_CODE	VRAM table code (Refer to Table 3-1) to be written.
START_INDEX	START_INDEX is a two-byte number which indicates the starting entry number of the table. For other considerations, refer to the START_INDEX parameter of GET_VRAM in Section 3.2.1.2.

1 DATA

2 Starting address of a data buffer
3 where data to be written to VRAM
4 resides.

5 COUNT

6 Number of entries to be put to the
7 VRAM table.

8 The restrictions on COUNT are
9 again table dependent. In other
10 words, it should always be the
11 case that $START\_INDEX + COUNT \leq$
12 Table Size.

13
14 Side Effects:

- 15
- 16 - Destroys AF, BC, DE, HL, IX and IY.
 - 17 - Uses local storage locations, SAVE_TEMP and
 - 18 SAVED_COUNT.
- 19

20 Calls to other OS routines:

- 21
- 22 - WRITE_VRAM
- 23
24
25
26

3.2.2 Table-Oriented Graphics Routines

A number of routines are included in the table level graphics software that perform useful operations on generators. Each of these takes a table code, a source index from that table, a destination index in the same table, and the number of entries to be processed. The routines work in read-modify-write mode, that is, they pull the generators out of the table one at a time, process them and put them back. They use a CRAM buffer for their scratch area. This buffer is allocated by the applications programmer and accessible only through the pointer at WORK_BUFFER in cartridge ROM.

With one exception, the routines in this package always process generators one at a time, and write them to the destination block in the same order in which they are extracted from the source block. This has important implications for their use with size 1 sprites.

When the sprite size is 1, the hardware accesses four generators at the index found in a sprite's attribute

1 table entry and displays them so that they appear on the
2 screen as shown in Figure 3-3.
3

4 Sprite Screen Location *

5 first generator	third generator
6 second generator	fourth generator

7
8
9 Figure 3-3
10 Sprite Size 1 Orientation
11

12 Thus, OS routines operating on the individual generators
13 for a size 1 sprite will not be sufficient to orient the
14 entire object. The four generators that make up the
15 sprite will have to be permuted as well. The
16 applications program will have to include a small
17 routine that performs the required permutation in tandem
18 with the OS call.
19

20 The following operations are available in the table-
21 oriented graphics package:
22
23
24
25
26

-
- | | |
|----|--|
| 1 | - Reflection about the vertical axis |
| 2 | - Reflection about the horizontal axis |
| 3 | - 90-degree rotation |
| 4 | - Enlargement by a factor of two |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| 16 | |
| 17 | |
| 18 | |
| 19 | |
| 20 | |
| 21 | |
| 22 | |
| 23 | |
| 24 | |
| 25 | |
| 26 | |

3.2.2.1 REFLECT_VERTICAL

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, SOURCE
LD    HL, DESTINATION
LD    BC, COUNT
CALL  REFLECT_VERTICAL
```

Description:

REFLECT_VERTICAL takes each generator in a block of COUNT generators following SOURCE in the table indicated by TABLE_CODE and modifies it in such a way that the new generator thus created will appear to be a reflection about the vertical screen axis of the old. The created generators are put back into a block of COUNT generators following DESTINATION in the same table.

The user must provide the permutation for size 1 sprite generators as diagrammed in Figure 3-4 below:

Block indicated by sprite name:•

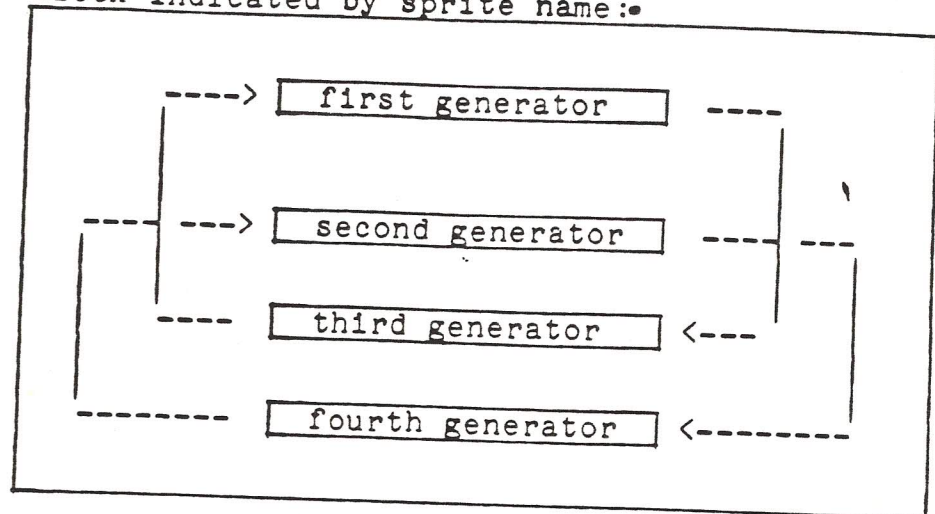


Figure 3-4
REFLECT_VERTICAL Size 1 Sprite Permutation

If TABLE_CODE is 3 (indicating the pattern generator table) and graphics mode 2 is used, REFLECT_VERTICAL also copies the color table entries for each generator it processes. Thus, when it is complete, the two-color table blocks indexed by SOURCE and DESTINATION will be identical. This means that the color scheme for the reflected generators will be the same as that for the originals.

Parameters:

TABLE_CODE

VRAM table code (Ref. Table 3-1)
to be operated upon.

SOURCE

SOURCE is the two-byte index of
the first entry in the specified
table to be operated on.

For table operations of sprite
generator or pattern generator in
graphics mode 1, SOURCE should be
in the range $0 \leq \text{SOURCE} \leq 255$.
For pattern generators in mode 2,
it should be in the range $0 \leq$
 $\text{SOURCE} \leq 767$. In either case, if
a value of SOURCE supplied is
outside the table's range but
still is a legal VRAM address, the
specified number of "entries" will
be read and modified from the VRAM
location (table location) + 8 *

1 SOURCE. For the proper table
2 entries and table boundary, refer
3 to Table 3-2.
4

5 Sprite size has no effect on the
6 range of SOURCE.
7

8
9
10 DESTINATION (HL) DESTINATION indexes the place where
11 REFLECT_VERTICAL will start putting
12 generators back into VRAM after
13 modifying them.
14

15 The same restrictions apply to the
16 value of DESTINATION as to the value of
17 SOURCE. They are both intended to be
18 indices into the same generator table.
19

20 COUNT (BC) A two-bytes count of the number of
21 entries to be processed sequentially
22 after SOURCE.
23
24
25
26

The legal value for COUNT is dependent on the size of the table being operated on and the values of SOURCE and DESTINATION. In general, both of the following statements should be true:

$COUNT + SOURCE \leq (\text{table size})$

$COUNT + DESTINATION \leq (\text{table size})$

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL', IX and IY.
- Uses the first 16 bytes of the data area pointed to by WORK_BUFFER.

Calls to other OS routines:

- GET_VRAM
- PUT_VRAM

3.2.2.2 REFLECT_HORIZONTAL

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, SOURCE
LD    HL, DESTINATION
LD    BC, COUNT
CALL  REFLECT_HORIZONTAL
```

Description:

REFLECT_HORIZONTAL takes each generator in a block of COUNT generators following SOURCE in the table indicated by TABLE_CODE and modifies it in such a way that the new generator created will appear to be a reflection about the horizontal screen axis of the old. The created generators are placed back into a block of COUNT generators following DESTINATION in the same table.

The user has to provide the permutation for size 1 sprite generators as diagrammed in Figure 3-5.

Block indicated by sprite name:

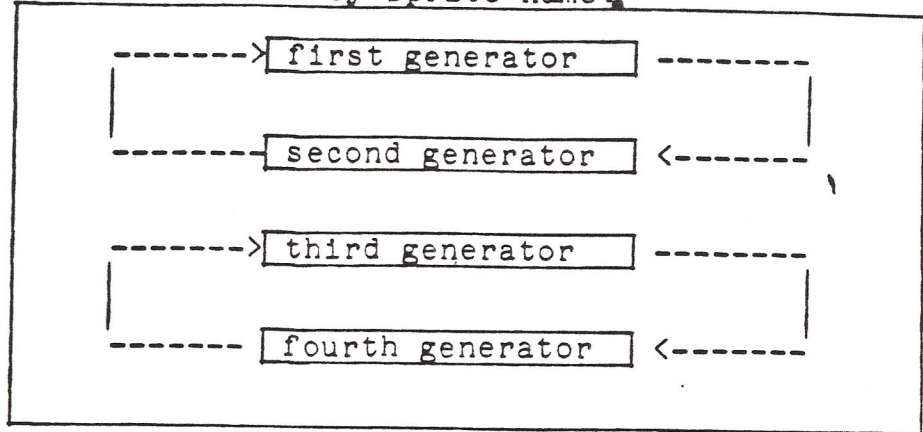


Figure 3-5
REFLECT_HORIZONTAL Size 1 Sprite Permutation

If TABLE_CODE is 3 (indicating the pattern generator table) and the graphics mode is 2, REFLECT_HORIZONTAL also performs the identical reflection on the corresponding color table entry for each generator it processes. This means that the reflected generators will be colored in a way that is consistent with their unreflected counterparts. When in mode 1, the color table is untouched.

Parameters:

TABLE_CODE

VRAM table code (Ref. Table 3-1)
to be operated upon.

SOURCE

SOURCE is the two-byte index of
the first entry in the specified
table to be operated on.

For table operations on sprite
generator or pattern generator in
graphics mode 1, SOURCE should be
in the range $0 \leq \text{SOURCE} \leq 255$.
For pattern generators in mode 2,
it should be in the range $0 \leq$
 $\text{SOURCE} \leq 767$. In either case, if
a value of SOURCE is supplied and
is outside the table's range but
still a legal VRAM address, the
specified number of "entries" will
be read and modified from the VRAM
location (table location) + $8 * \text{SOURCE}$. For the proper table
entries and table boundary, refer
to Table 3-2.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

Sprite size has no effect on the range of SOURCE.

DESTINATION

DESTINATION indexes the place where REFLECT_VERTICAL will start putting generators back into VRAM after modification.

The same restrictions apply to the value of DESTINATION as to the value of SOURCE. They are both intended to be indices into the same generator table.

COUNT

A two-byte count of the number of entries to be processed sequentially after SOURCE.

A legal value for count depends on the size of the table being operated on and the values of SOURCE and DESTINATION. In

general, both of the following
statements should be true:

COUNT + SOURCE <= (table size)
COUNT + DESTINATION <= (table
size)

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL', IX and IY.
- Uses the first 16 bytes of the data area pointed to by
WORK_BUFFER.

Calls to other OS routines:

- GET_VRAM
- PUT_VRAM

3.2.2.3 ROTATE_90

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, SOURCE
LD    HL, DESTINATION
LD    BC, COUNT
CALL  ROTATE_90
```

Description:

ROTATE_90 takes each generator in a block of COUNT generators following SOURCE in the table indicated by TABLE_CODE and modifies it in such a way that the new generator thus created will appear to be a 90-degree clockwise rotation of the old. The created generators are put back into a block of COUNT generators following DESTINATION in the same table.

The user must provide the permutation for size 1 sprite generators as diagrammed in Figure 3-6 below:

Block indicated by sprite name:

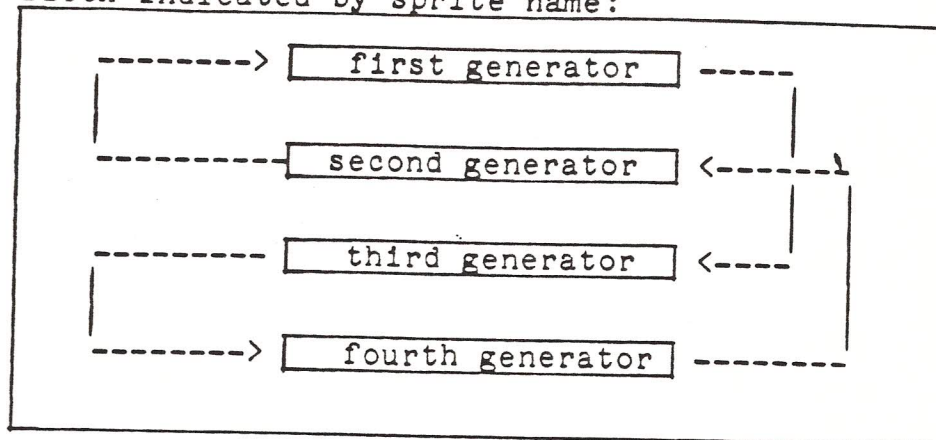


Figure 3-6
ROTATE_90 Size 1 Sprite Permutation

This routine should be used with great care when applied to pattern generators in mode 2. In this mode, the VDP allows arbitrary color combinations along vertical lines while it is still limited to two colors along a given 8-pixel horizontal line. The problem is that if the user attempts to rotate a figure that has more than two colors on a vertical line, ROTATE_90 will exhibit color problems after rotation. There is no way around this problem except to keep any generators that are intended for rotation simple. If the TABLE_CODE is 3 (pattern

1 generator table) and the mode is 2, ROTATE_90 will copy
2 the corresponding color table entries indexed by SOURCE
3 to the block indexed by DESTINATION.
4

5
6 Parameters:

7
8 TABLE_CODE VRAM table code (Ref. Table 3-1)
9 to be operated upon.

10
11 SOURCE SOURCE is the two-byte index of
12 the first entry in the specified
13 table to be operated on.
14

15 For table operations of sprite
16 generator or pattern generator in
17 graphics mode 1, SOURCE should be
18 in the range $0 \leq \text{SOURCE} \leq 255$.
19 For pattern generators in mode 2,
20 it should be in the range $0 \leq$
21 $\text{SOURCE} \leq 767$. In either case, if
22 a value of SOURCE is supplied and
23 is outside the table's range but
24
25
26

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

still is a legal VRAM address, the specified number of "entries" will be read and modified from the VRAM location (table location), + 8 * SOURCE. For the proper table entries and table boundary, refer to Table 3-2.

Sprite size has no effect on the range of SOURCE.

DESTINATION

DESTINATION indexes the place where REFLECT_VERTICAL will start putting generators back into VRAM after modifying them.

The same restrictions apply to the value of DESTINATION as to the value of SOURCE. They are both intended to be indices into the same generator table.

COUNT

A two-byte count of the number of entries to be processed sequentially after SOURCE.

The legal value for count is dependent on the size of the table being operated on and the values of SOURCE and DESTINATION. In general, both of the following statements should be true:

$COUNT + SOURCE \leq (\text{table size})$
 $COUNT + DESTINATION \leq (\text{table size})$

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL' IX and IY.
- Uses the first 16 bytes of the data area pointed to by WORK_BUFFER.

1

2

Calls to other OS routines:

3

- GET_VRAM

4

- PUT_VRAM

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

1

2

3

3.2.2.4 ENLARGE

4

5

Calling Sequence:

6

LD A, TABLE_CODE

7

LD DE, SOURCE

8

LD HL, DESTINATION

9

LD BC, COUNT

10

CALL ENLARGE

11

12

Description:

13

14

ENLARGE takes each generator in a block of COUNT
generators following SOURCE in the table indicated by
TABLE_CODE and from it creates four generators as shown
below in Figure 3-7.

15

16

17

18

19

20

21

22

23

24

25

26

first generator	third generator
second generator	fourth generator

Figure 3-7
ENLARGE Generators Layout

The enlarged object will appear to be a double-sized version of the original. The created generators are put back into a block of 4 * COUNT generators following DESTINATION in the same table.

Note that since the ordering of the expanded generators is the same as that for the four generators needed to produce a size 1 sprite, ENLARGE lends itself well to use with sprites as long as the programmer is willing to dedicate four times as many sprites to the expanded object as to the original.

If TABLE_CODE is 3 (indicating the pattern generator table) and the graphics mode is 2, ENLARGE makes four

1 copies of the color table entry for each source
2 generator and places them in the color table so that
3 they correspond to the four destination generators.
4 This should mean that the color scheme for the enlarged
5 object will be the same as that of the original. If the
6 mode is 1, the color table is untouched.

7
8 Parameters:

9
10 TABLE_CODE VRAM table code (Ref. Table 3-1)
11 to be operated upon.

12
13 SOURCE SOURCE is the two-byte index of
14 the first entry in the specified
15 table to be operated on.

16
17 For table operations on a sprite
18 generator or a pattern generator
19 in graphics mode 1, SOURCE should
20 be in the range $0 \leq \text{SOURCE} \leq$
21 255. For pattern generators in
22 mode 2, it should be in the range
23
24
25
26

1
2 0 <= SOURCE <= 767. In either
3 case, if a value of SOURCE is
4 supplied and is outside the
5 table's range but still a legal
6 VRAM address, the specified number
7 of "entries" will be read and
8 modified from the VRAM location
9 (table location) + 8 * SOURCE.
10 For the proper table entries and
11 table boundary, refer to Table
12 3-2.

13
14 Sprite size has no effect on the
15 range of SOURCE.

16
17 DESTINATION

18 DESTINATION indexes the place
19 where ENLARGE will start placing
20 generators back into VRAM after
21 modifying them.

22 The same restrictions apply to the
23 value of DESTINATION as to the
24
25
26

1

value of SOURCE. They are both intended to be indices into the same generator table.

2

3

4

5

COUNT

6

A two-byte count of the number of entries to be processed sequentially after SOURCE.

7

8

9

10

The most important factor limiting the size of COUNT in the case of the ENLARGE routine is that ENLARGE actually produces four generators for every generator that it reads.

11

12

13

14

15

16

17

18

19

20

The legal value for count depends on the size of the table being operated on and the values of SOURCE and DESTINATION. Both of the following statements should be true:

21

22

23

24

25

26

COUNT + SOURCE <= (table size)
DESTINATION + 4 * COUNT <=
(table size)

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL', IX and IY.
- Uses the first 40 bytes of the data area pointed to by WORK_BUFFER.

Calls to other OS routines:

- GET_VRAM
- PUT_VRAM

3.2.3 Sprite Reordering Software

Probably the most significant hardware limitation of the VDP is the so-called "fifth sprite problem." This problem arises when more than four sprites occur on a single horizontal scan line. Because the chip only has four registers for dealing with the lower order sprites, the sprites with the higher sprite attribute indices cannot be generated on that scan line and therefore disappear.

One solution to this problem is to use a reordering scheme on the offending sprites which involves swapping the priorities of the sprite that is being blanked out with that of one of the higher order sprites in the group on successive video fields. The result is that while the sprites that are being reordered tend to flicker in the area of overlap, they are still quite visible. The degree of flicker depends on many factors including the color of the sprites in question and the background color and complexity.

1 The OS supports this solution by allowing the
2 application to adjust the order of sprite attribute
3 entries with minimum effort.

4
5 Two tables are used in implementing the sprite
6 reordering feature. The first of these is simply a
7 local CRAM version of the VRAM sprite attribute table.
8 It must be allocated by the application program and made
9 accessible to the OS by placing a pointer to it at the
10 predetermined cartridge ROM location LOCAL_SPR_TBL.
11 This local sprite attribute table need only contain the
12 active sprite entries needed by the application and
13 therefore may be shorter than the 128 bytes required for
14 the VRAM version. The other table is called the sprite
15 order table. It is also allocated by the application
16 program through a pointer, SPRITE_ORDER, located in
17 cartridge ROM. The sprite order table should contain
18 one byte for each entry in the local sprite attribute
19 table, and the bytes should take on values in the range
20 $0 \leq b \leq 31$.

21
22 When the flag MUX_SPRITES is false (0), PUT_VRAM writes
23 sprite attribute entries directly to VRAM. However,
24
25
26

1 when this flag becomes true (1), they are written
2 instead to the local sprite attribute table. Then, a
3 routine called WR_SPR_NM_TBL will map the local sprite
4 attribute entries to VRAM according to the sprite order
5 table.
6

7 An example of the relationship between the three tables
8 may be illustrated as follows:
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

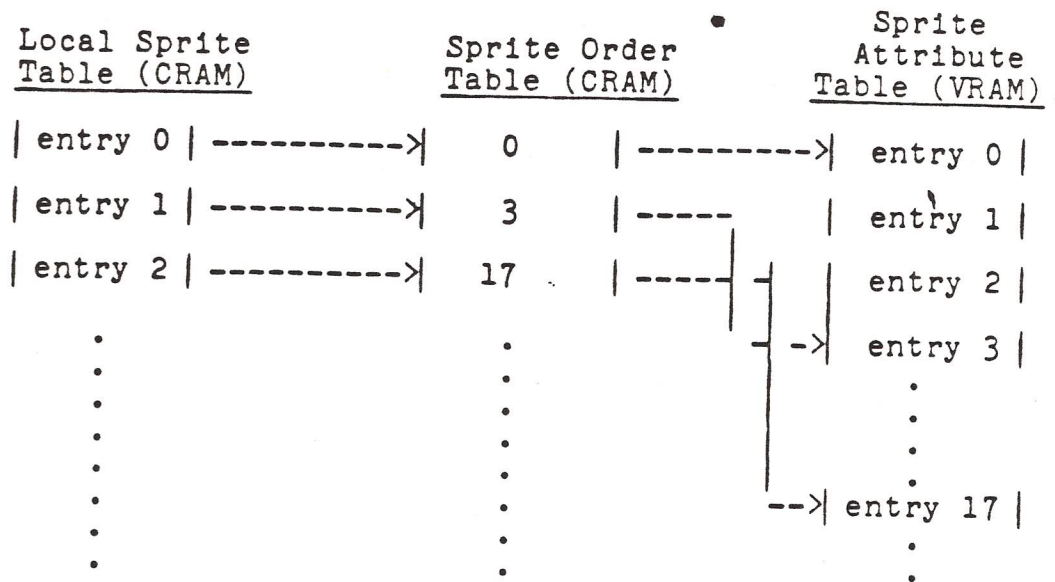


Figure 3-8
 Sprite Reordering Table Mapping

The advantage of this method lies in the fact that it takes a lot less work to reorder the bytes in the sprite order table than it does to move around the entries in the VRAM or CRAM sprite attribute tables.

3.2.3.1 INIT_SPR_ORDER

Calling Sequence:

```
LD    A, SPRITE_COUNT
CALL  INIT_SPR_ORDER
```

Description:

INIT_SPR_ORDER looks at the pointer SPRITE_ORDER in low cartridge ROM which should contain the address of a free area SPRITE_COUNT bytes long in CRAM. It sets this area up as a sprite order table by initializing it with zero through SPRITE_COUNT - 1.

Parameters:

SPRITE_COUNT	The length of the sprite order table, which should be the same as the intended number of entries in the local sprite attribute table.
--------------	---

1

This number must always be in the
range 1 <= SPRITE_COUNT <= 32.

2

3

4

Side Effects:

5

6

- Destroys AF, BC, and HL.

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

3.2.3.2 WR_SPR_NM_TBL

Calling Sequence:

```
LD    A, COUNT
CALL  WR_SPR_NM_TBL
```

Description:

WR_SPR_NM_TBL writes COUNT entries from the local sprite attribute table, which it accesses through the pointer LOCAL_SPR_TBL in low cartridge ROM, to the VRAM sprite attribute table. The transfer is mapped through the sprite order table which it accesses through the pointer SPRITE_ORDER in low cartridge ROM.

Parameters:

COUNT	This is the number of sprite attribute entries to be written to VRAM.
-------	---

COUNT should not be larger than
the initialized length of the
sprite order table.

Side Effects:

- Destroys AF, BC, DE, HL, IX and IY.
- Cancels any previously established VDP operations.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

3.3 Object Level

The object level software constitutes the top level of the graphics generation software, which appears to the user as a collection of screen objects with well-defined shape, color scheme, and location at any given moment. The software supports four distinct object types, each of which has its own capabilities and limitations. Once objects are defined, however, the rules for manipulating them are fairly type-independent. In fact, only one routine (PUTOBJ), is used to display objects of all types.

Brief descriptions are given in the following sections in regard to object types, object data structures and two user-accessible routines (ACTIVATE, PUTOBJ). For further information, refer to Appendix B.

3.3.1 Object Types

There are four different types of objects defined by the OS. A brief description for each type is given below.

1 3.3.1.1 Semi-Mobile
2
3

4 Semi-mobile objects are rectangular arrays of pattern
5 blocks which are always aligned on pattern boundaries.
6 Their animation capability is limited. In most cases
7 they are used to set up background pattern graphics.

8 3.3.1.2 Mobile
9

10 The size of a mobile object is fixed in two-by-two
11 pattern blocks. They belong to the pattern plane but
12 can be moved from pixel to pixel in X,Y directions like
13 a sprite superimposed on the background. However, the
14 speed of mobile objects are too slow when compared to
15 the sprites.
16

17 3.3.1.3 Sprite
18

19 Sprite objects are composed of an individual sprite.
20
21
22
23
24
25
26

1
2 3.3.1.4 Complex

3
4 Complex objects are collections of other "component"
5 objects which may be of any type including other complex
6 objects.
7

8
9 3.3.2 Object Data Structure

10 Each of the above mentioned objects has its definition
11 in cartridge ROM. This high-level definition links
12 together several different data areas which specify all
13 aspects of an object. The data structure is described
14 in detail in Appendix B.
15

16
17 3.3.2.1 Graphics Data Area

18
19 This data area is located in cartridge ROM. Pattern and
20 color generators for semi-mobile, mobile and sprite
21 objects and frame data for all objects are located in
22 the graphics data area. The data structure within each
23 graphics area depends on the type of object with which
24 it is associated. If, however, two or more objects of
25 the same type are graphically identical, they may share
26

1 the same graphics area. This will reduce the amount of
2 graphics data that needs to be stored in cartridge ROM.
3

4 3.3.2.2 Status Area
5

6 Each object will have its own status area in CRAM. The
7 game program uses this area to manipulate the object.
8 It does this by altering the location within status
9 which determines which frame is to be displayed as well
10 as the locations which define the position of the object
11 on the display. The graphics routine, PUTOBJ, when
12 called, will access the object's status area and place
13 the object accordingly.
14

15 3.3.2.3 OLD_SCREEN
16

17 Mobile and semi-mobile objects appear in the pattern
18 plane. They are displayed by altering some of the names
19 in the pattern name table. The original names represent
20 a background which is "underneath" the object. When the
21 object moves or is removed from the pattern plane, the
22 original names must be restored to the name table.
23
24
25
26

1 Before placing a semi-mobile or mobile object on the
2 display, PUTOBJ will restore any previously saved names
3 and also save the names which constitute the background
4 underneath the new location of the object. Sprite and
5 complex objects do not need OLD_SCREEN areas.
6

7 3.3.3 ACTIVATE
8

9 Calling Sequence:
10

11 LD HL, OBJ_DEF

12 SCF

13 CALL ACTIVATE
14

15 or
16

17 LD HL, OBJ_DEF

18 OR A

19 CALL ACTIVATE
20
21
22
23
24
25
26

1
2 Description:

3
4 The primary purpose of this routine is to move the pat-
5 tern and color generators from the graphics data area
6 into the pattern and color generator tables in VRAM.
7 Each object must be "activated" before it can be
8 displayed. ACTIVATE also initializes the first byte in
9 an object's OLD_SCREEN data area with the value 80H.
10 PUTOBJ tests this location before restoring the
11 background names to the name table. If the value 80H is
12 found, it is an indication that there are no background
13 names to restore.

14
15 Parameters:

16 OBJ_DEF High level definition of an
17 object. See Appendix B for
18 further details.

19
20 SCF Carry flag should be set if user
21 wishes to load the generators spe-
22 cified for this object.
23
24
25
26

OR A

Carry flag should be reset if user
knows that the generators are
already in VRAM.

3.3.4 PUTOBJ

Calling Sequence:

```
LD    IX, OBJ_DEF
LD    B, BKGND_SELECT
CALL  PUTOBJ
```

Description:

PUTOBJ is called when an object's frame or its
location on the display is to be changed. The routine
tests the type of object and then branches to one of
several subroutines designed to handle that particular
object type. These routines are not accessible to the
user. Their functions are as follows:

1. PUT_SEMI

Semi-mobile objects are placed on the display by writing the generator names specified by one of the object's frames into the pattern name table in VRAM. The pattern and color generators which are needed to create the frame must already be in their respective generator tables.

2. PUT_MOBILE

Mobile objects are displayed by producing a new set of pattern and color generators which depict the frame to be displayed on the background. These new generators are then moved to the locations in the VRAM pattern and color generator tables which are reserved for the object; the names of the new generators are then written into the pattern name ta

3. PUT_SPRITE0

PUT_SPRITE0 handles the display of size 0 sprite objects.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

4. PUT_SPRITE1

PUT_SPRITE1 handles the display of size 1 sprite objects.

5. PUT_COMPLEX

PUT_COMPLEX calls PUTOBJ for each of its component objects.

Parameters:

OBJ_DEF

High level definition of an object. See Appendix B for further details.

BCKGND_SELECT

Used with mobile objects or complex objects with a mobile-type component. Can be ignored otherwise. For methods of selecting background colors in a mobile object. Refer to Appendix B for additional information.

SECTION IV

INTERRUPT HANDLING AND WRITE DEFERRAL

The 60Hz (50Hz for European version) non-maskable interrupt (NMI) in the ColecoVision system has a wide variety of applications such as providing a fixed time base for the timing software, and a natural debounce interval for the controller interface. However, interrupts can cause problems if not handled properly.

Let us say, for example, that the system is in the midst of a call to PUTOBJ and is, in fact, writing to VRAM when the interrupt occurs. If the interrupt service routine calls for transferring data to another area of VRAM by setting up the VDP address register (auto-increment) to a different value, the pending VRAM operation cannot resume properly after the interrupt is serviced.

The OS contains software which allows graphics operations on the object level to be protected against damage by asynchronous interrupts. It should be stressed that the OS protects ONLY the

1 object level. Routines on the table and chip driver levels could
2 be deferred against interrupt by using the application library
3 routine, DEF_INT, suggested in ColecoVision Bulletin No. 0010
4 (Appendix D).
5

6 In order to implement this protection for graphics operations,
7 the application program must allocate space for a deferral queue.
8 The size of this queue depends on the number of graphics
9 operations that are expected to be performed between NMIs, but
10 usually fifteen entries of three bytes each will prove
11 sufficient. The address of the queue should be passed on to the
12 OS using a routine called INIT_WRITER which also empties the
13 queue and prepares it for operation. Thereafter, whenever the
14 flag byte DEFER_WRITES is set to true (1), calls to PUTOBJ are
15 deferred by placing them on the queue where they may be performed
16 using a single OS call from the interrupt service routine.
17

18 In addition to the buffer in which the queue resides, the
19 deferral routines use several defined storage areas in the course
20 of their operation. These are: QUEUE_SIZE, QUEUE_HEAD,
21 QUEUE_TAIL, HEAD_ADDRESS, TAIL_ADDRESS and BUFFER. They are all
22 related to the state of the queue.
23
24
25
26

4.1 INIT_WRITER

Calling Sequence:

```
LD    A, SIZE
LD    HL, LOCATION
CALL  INIT_WRITER
```

Description:

INIT_WRITER initializes the queue. It does not, in fact, do anything to the "physical" queue in RAM. Instead, it merely sets up its description by setting QUEUE_SIZE to SIZE, HEAD_ADDRESS and TAIL_ADDRESS to the beginning of the buffer and QUEUE_HEAD and QUEUE_TAIL to 0.

Parameters:

SIZE The size in entries of the queue.
SIZE should be equal to the amount

of space allocated for the queue
divided by three. Range for SIZE
is 1 to 255.

LOCATION

The location of CRAM area
allocated for the queue.

Side Effects:

- Destroys AF.

1

2

3

4.2 WRITER

4

5

Calling Sequence:

6

7

CALL WRITER

8

9

Description:

10

11

WRITER performs any deferred PUTOBJ operations that may be on the queue emptying the queue as it goes. WRITER should be called by the interrupt service routine.

12

13

14

15

WRITER uses a "back door" into the PUTOBJ software without ever making an explicit call to PUTOBJ.

16

17

18

Side Effects:

19

20

Destroys AF, BC, DE, HL, IX, IY, BC', DE' and HL'.

21

22

23

24

25

26

SECTION V
TIMING SOFTWARE

Timing software enables the user to specify a preset length of time and to signal the user when that time has elapsed. In theory, up to 255 software timers are available to the user.

The Z80-CPU's non-maskable interrupt (NMI) input, which comes from the VDP interrupt output, forms the time base for all the timers. In the U.S., it is about every 1/60 second. In the European version, it is about every 1/50 second. `TIME_MGR` is the routine responsible for generating the time base at the desired intervals.

All timer routines use a CRAM area designated as the `TIMER_TABLE`. The size of this table depends on the number of timers in use and their types. There are two types of timers, non-repeating and repeating.

The user will be notified of the status of the timer only when he checks it.

1 5.1 Non-Repeating Timers

2
3 These timers will not repeat themselves after time out.
4 The user will be notified and their timers are set free.
5

6 5.2 Repeating Timers

7
8 These timers only need to be set once. After each time
9 out they will notify the user and repeat themselves.
10

11 For both types of timers, the timer length can be either
12 short or long:

13
14 (a) Short - 1 to 255 units of time base.

15 (b) Long - 256 to 65535 units of time base.
16

17 5.3 TIMER_TABLE:

18
19 As a timer is requested, it is placed into TIMER_TABLE.
20 Each timer consists of a Mode_Byte and a two-byte
21 Value_Word.
22
23
24
25
26

The appropriate amount of CRAM should be reserved using the following formula:

TIMER_TABLE	DEFS	Num_of_Timers * 3
TIMER_DATA_BLOCK	DEFS	Num_of_Long_Repeating * 4

NOTE: Num_of_Timers is the total number of timers.

5.3.1 Mode_Byte

Mode_Byte is one byte of data for each timer containing the information of done bit, repeat bit, free bit, long bit and last-timer-in-table bit (Refer to Appendix G for details).

5.3.2 Value_Word

A two-byte value which can have several meanings depending on the type of timer.

(a) Short Timers:

The Value_Word is the actual timer in this case.

1 The first byte is the value to be decremented and
2 the second byte is the initial timer value. In the
3 case of a repeating timer, the second byte is used
4 to restart the timer.
5

6 (b) Long Non-Repeating Timers:
7

8 The Value_Word is the value of the timer and is
9 decremented as a two-byte quantity.
10

11 (c) Long Repeating Timers:
12

13 The Value_Word is the address of the location in
14 the TIMER_DATA_BLOCK where the first word is the
15 value to be decremented and the second word is the
16 initial timer value.

17 5.3.3 TIMER_DATA_BLOCK
18

19 This is the data area in CRAM where four bytes are
20 designated for each long repeating timer. The table's
21 size is expandable under user control, so care should be
22 taken not to write over other pertinent data.
23
24
25
26

1

2

3

5.4 INIT_TIMER

4

5

Calling Sequence:

6

7

LD HL, TIMER_TABLE

8

LD DE, TIMER_DATA_BLOCK

9

CALL INIT_TIMER

10

11

Description:

12

13

INIT_TIMER initializes timer data areas to the locations defined as inputs.

14

15

16

Parameters:

17

18

TIMER_TABLE

This is the CRAM address where the timer information will be placed.

19

20

21

22

TIMER_DATA_BLOCK

This is the address where long repeating timer data will be placed.

23

24

25

26

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

Side Effects:

- Destroys DE and HL.

5.5 TIME_MGR

Calling Sequence:

CALL TIME_MGR

Description:

TIME_MGR is responsible for maintaining all OS software timers. The task of maintenance is defined as updating all timers, setting the proper signal code when a timer times out, and restarting repeating timers.

Each call to TIME_MGR will cause all active timers to be decremented by one. There is no limit as to when the routine could be called, but typically it is every NMI from VDP which forms the system time base.

An active timer is defined as a timer with its repeat bit set or its done bit not set, or both.

1 If an entire timer value decrements to zero, the done
2 bit will be set in Mode_Byte. In addition, the timer
3 will be restarted if it is a repeating type.
4

5 Parameters: None.
6

7 Side Effects:
8

9 - Destroys AF, DE and HL.
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

5.6 REQUEST_SIGNAL

Calling Sequence:

```
LD    HL, TIMER_LENGTH
LD    A, REPEAT
CALL  REQUEST_SIGNAL
```

Description:

REQUEST_SIGNAL accepts a time interval and a repeat code (Boolean) as input. The REPEAT parameter, when set, instructs TIME_MGR to re-initialize the timer when it times out instead of relinquishing the timer memory locations.

REQUEST_SIGNAL sets up a timer and assigns that timer a number in the accumulator. The routine also determines the type of timer and allocates space in the TIMER_TABLE accordingly.

Short Timer:

A short timer has a counter value of 255 or less and uses one Mode_Byte and Value_Word.

Long Timer:

A long timer has counter values greater than 255.

(a) Non-Repeating:

A non-repeating timer uses a Mode_Byte and Value_Word.

(b) Repeating:

A repeating timers uses a Mode_Byte and Value_Word in addition to four bytes starting at the first available location in the TIMER_DATA_BLOCK.

The user should save the timer number. This value, referred to as SIGNAL_NUM, should subsequently be used when calling TEST_SIGNAL to find the status of the signal or when calling FREE_SIGNAL to release a timer.

Parameters:

TIMER_LENGTH

The number of the time base units
of a timer. Values range from 1
(shortest) to OFFFFH (longest).

REPEAT

1 = repeating timer;
0 = non-repeating timer.

Output:

Value of timer number returned in
accumulator. User should save it
in CRAM location SIGNAL_NUM.

Side Effects:

- Destroys AF, BC, DE, and HL.

5.7 TEST_SIGNAL

Calling Sequence:

```
LD    A, SIGNAL_NUM
CALL  TEST_SIGNAL
```

Description:

TEST_SIGNAL takes a signal number and tests to see whether the indicated timer has timed out since the last time it was tested. If so, it returns with "true" in the accumulator; otherwise, it returns "false". The zero flag reflects the contents of the accumulator.

If the timer of SIGNAL_NUM tested has its Done bit set and the timer is non-repetitive, then the Free bit will be set to release the timer for further use.

If no timer of a particular signal number exists then the routine will return a false.

1 Although it has been defined that testing a non-existing
2 signal number will return a false value, a common error
3 in use of timing routines is the testing of an undefined
4 signal.

5
6 The error occurs when one module, with a given
7 SIGNAL_NUM, calls TEST_SIGNAL with that SIGNAL_NUM as
8 input. If this module receives a "true" from
9 TEST_SIGNAL, then another module which is rightfully
10 using a timer with that SIGNAL_NUM will not receive
11 a "done" signal.

12
13 Parameters:

14
15 SIGNAL_NUM Timer number.

16
17 Side Effects:

18
19 - Destroys AF (output), BC, DE, and HL.
20
21
22
23
24
25
26

5.8 FREE_SIGNAL

Calling Sequence:

```
LD    A, SIGNAL_NUM  
CALL FREE_SIGNAL
```

Description:

FREE_SIGNAL takes a SIGNAL_NUM value as input and upon finding a timer assigned to that number, releases it to the free list. If no timer of that SIGNAL_NUM is found, no action will be taken. This routine will free a timer regardless of its current value or its REPEAT parameter.

Special case of long repeating timer:

This routine will release a portion of the TIMER_DATA_BLOCK that a particular timer uses and moves the remaining contents up. The Value_Words of other

1 repeating timers will also be modified to reflect this
2 move.

3
4 NOTE: In this special case, TIME_MGR, or any other
5 routine that accesses or modifies the
6 TIMER_TABLE, should NOT be called during the exe-
7 cution of FREE_SIGNAL. (This may occur if
8 TIME_MGR was called on interrupt). ColecoVision
9 Bulletin No. 0010 (Appendix D) suggests the
10 solution of using DEF_INT to defer interrupts.

11
12
13 Parameters:

14
15 SIGNAL_NUM Previously defined output from
16 REQUEST_SIGNAL.

17
18 Side Effects:

19
20 - Destroys AF, BC, DE and HL.
21
22
23
24
25
26

1 repeating timers will also be modified to reflect this
2 move.

3
4 NOTE: In this special case, TIME_MGR, or any other
5 routine that accesses or modifies the
6 TIMER_TABLE, should NOT be called during the exe-
7 cution of FREE_SIGNAL. (This may occur if
8 TIME_MGR was called on interrupt). ColecoVision
9 Bulletin No. 0010 (Appendix D) suggests the
10 solution of using DEF_INT to defer interrupts.
11

12 Parameters:

13
14 SIGNAL_NUM Previously defined output from
15 REQUEST_SIGNAL.
16

17 Side Effects:

18
19 - Destroys AF, BC, DE and HL.
20
21
22
23
24
25
26

SECTION VI
CONTROLLER INTERFACE

Most applications involving the hand controller require similar needs in decoding and debouncing those inputs. The operating system addresses those needs in one general purpose routine, POLLER. POLLER will decode and debounce either all or selected portions of the hand controller hardware and place the processed data in the Controller Data Area selected by the pointer in CONTROLLER_MAP.

Special applications may require non-standard decoding of the inputs available from the hardware; therefore, entry points to lower level routines are available.

There are four routines available to access controller inputs:

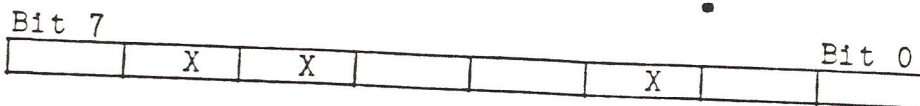
- POLLER
- DECODER
- CONT_SCAN
- UPDATE_SPINNER

6.1 Controller Data Area

The pointer in CONTROLLER_MAP points to the user-defined CRAM area which is accessed and/or modified when POLLER is called. Users define this address by placing the location of the 12 bytes of the CRAM Controller Data Area at cartridge location CONTROLLER_MAP. They are defined as follows:

+0	Player 1 enable	
+1	Player 2 enable	
+2	Fire button (left button)	Player 1
+3	Joystick	Player 1
+4	Spinner count (for interface modules)	Player 1
+5	Arm button (right button)	Player 1
+6	Keyboard	Player 1
+7	Fire button	Player 2
+8	Joystick	Player 2
+9	Spinner count	Player 2
+10	Arm button	Player 2
+11	Keyboard	Player 2

Player Enable (+0, +1):



Where bit = 1: Function enabled.

bit = 0: Function disabled.

X = Don't care

While functions are as follows:

Bit 7 = Controller Enable

Bit 4 = Keypad

Bit 3 = Arm Button

Bit 1 = Joystick

Bit 0 = Fire Button

Status of individual portions of the controller map area
when enabled is described as follows:

Fire button:

Status = 040H, if fire button pressed

Status = 0H, if fire button not pressed

Joystick:

<u>Status</u>	<u>Direction</u>
01H	N
03H	NE
02H	E
06H	SE
04H	S
0CH	SW
08H	W
09H	NW

Spinner Switch:

SPIN_SW_CNT is added to the value for position offset.
(Ref to Sec. 6.5)

Arm Button:

Status = 0040H if arm button pressed

Status = 0000H if arm button not pressed

Keypad:

1		
2		
3		
4	<u>Value</u>	<u>Key</u>
5	00H	0
6	01H	1
7	02H	2
8	03H	3
9	04H	4
10	05H	5
11	06H	6
12	07H	7
13	08H	8
14	09H	9
15	0AH	*
16	0BH	#
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		

6.2 POLLER

Calling Sequence:

CALL POLLER

Description:

Reads, decodes and debounces all active portions of both controllers. The results are placed in the Controller Data Area.

POLLER's debounce algorithm waits until it finds the data the same for two successive passes before it modifies the Controller Data Area. If a particular portion is disabled, then this routine will still be looking for the second occurrence upon re-enabling. Please note that the POLLER routine cannot interrupt itself.

1

2

Side Effects:

3

4

-Destroys all except alternate register pairs, does not
destroy alternate AF pair.

5

6

- Zero's SPIN_SW_CNT if that portion of the controller
is enabled. (See UPDATE_SPINNER).

7

8

Calls to other OS routines:

9

10

- CONT_SCAN

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

0 = Fire, Joystick, Spinner

1 = Arm, Keypad

OUTPUTS:

IF SEGMENT CHOSEN WAS:

Segment 0

Segment 1

Register H

Fire

Arm

Register L

Joystick

Keyboard

Register E

Spinner

The decoded values are listed in the Controller Data Area.

Side Effects:

- Destroys AF, BC, DE and HL.

Calls to other OS routines:

- CONT_SCAN

6.4 CONT_SCAN

Calling Sequence:

CALL CONT_SCAN

Description:

Reads the actual ports to both controllers and places the data in an OS-defined CRAM area. These locations are labeled as SO_CO, SO_C1, S1_CO and S1_C1.

Side Effects:

- Destroys AF.

6.5 UPDATE_SPINNER

Calling Sequence:

```
ORG 801EH
JP UPDATE_SPINNER
```

Description:

For use with expansion modules only. Interrupt service routine which processes controller spinner switch interrupts (maskable). Decrements OS reserved byte SPIN_SW0_CNT for Controller No. 0 or SPIN_SW1_CNT for Controller No. 1 if spinner is going in one direction; increments byte if spinner is going in the other direction (Ref. Table 10-1).

NOTE: SPIN_SW_CNT is accessed and modified by both DECODER and POLLER if they are called.

SECTION VII
SOUND GENERATION SOFTWARE

The OS provides sound generation routines that output frequency, attenuation and control data to the TI SN76489 sound generator controller. The "sound" described in this section can be represented as both music or noise.

There is at least one ten-byte block of CRAM called a "Sound Data Area" reserved for each sound channel. This area contains a record of the current values "playing" on that sound channel. These values are the timing and descriptive information which generate musical notes that are originally stored in cartridge ROM. In total, there should be a minimum of four sound data areas reserved by the user, one for each channel. More data areas are needed if there are sounds to be played concurrently. For an average video game, seven is the required number.

Basically, in order to generate sound effects, the user has to prepare music notes and call the sound generation routines. The notes table, pointer and four routines are described below. For detailed information, refer to the Sound Users' Manual in Appendix C.

7.1 LST_OF_SND_ADDRS and PTR_TO_LST_OF_SND_ADDRS

All the music notes for an application program starts at the address called LST_OF_SND_ADDRS in cartridge ROM. There is another dedicated CRAM pointer located at address PTR_TO_LST_OF_SND_ADDRS which points to the LST_OF_SND_ADDRS. It is the user's responsibility to set up the pointer before passing control to any sound generation software.

7.2 SOUND_INIT

This routine should be called immediately after power on, before any sound processing can occur. It turns off the sound generators, initializes the CRAM locations to be used as sound data areas, sets up the four channel data area pointers and initializes PTR_TO_LST_OF_SND_ADDRS.

INPUT: n

TYPE: 8-bit constant

PASSED: in B

DESCRIPTION: Number of sound data areas used by the game.

1 INPUT: LST_OF_SND_ADDRS
2 TYPE: 16-bit address •
3 PASSED: in HL
4 DESCRIPTION: LST_OF_SND_ADDRS is the base
5 address of a list of the starting
6 addresses for each sound's note
7 list and data area.

8
9 OUTPUT: 1. Turns off all sound
10 generators.
11 2. Initializes
12 PTR_TO_LST_OF_SND_ADDRS.
13 3. Writes the inactive code
14 (OFFH) to byte 0 of the n
15 sound data areas.
16 4. Stores 00 at end of sound data
17 areas.
18 5. Sets the 4 channel sound
19 pointers to a dummy inactive
20 area.
21 6. Sets SAVE_CTRL to OFFH. (See
22 "Noise Notes" discussion in
23
24
25
26

ColecoVision Sound Users'
Manual in Appendix C).

7.3 PLAY_IT

PLAY_IT is called to start a sound. Using a sound number passed in B, PLAY_IT loads the data for the sound's first note into the appropriate sound data area, thereby truncating whatever sound had been "playing" in that data area. (The address of the appropriate area is found by using the sound number as an index into the LST OF_SND_ADDRS table). It also formats the data area's header and sets up the next note pointer. If the sound is a special sound effect, its next note pointer is set to the address of the special effect routine. The next time PLAY_SONGS is called, that sound's first note will be played.

If PLAY_IT is called with a sound number of a sound which is already in progress, it returns immediately (i.e., it doesn't restart the sound).

1 INPUT: Sound number to be started.

2 TYPE: 8-bit constant, 1 to 61.

3 PASSED: In B.

4 CALLS: PT_IX_TO_SxDATA,
5 LOAD_NEXT_NOTE PTR,
6 UP_CH_DATA_PTRS.
7

8 OUTPUT: 1. Moves the sound's first note
9 data to the appropriate sound
10 data area.
11 2. Formats byte 0 header of the
12 sound's data area.
13 3. Points next note pointer in
14 data area (bytes 1 & 2) to
15 address of first note in
16 sound, or address of special
17 sound effect routine.
18
19
20
21
22
23
24
25
26

7.4 SOUND_MAN

SOUND_MAN should be called every VDP interrupt. For each data area, SOUND_MAN processes the appropriate timer and sweep counters and modifies the frequency and attenuation data accordingly. If the data area is assigned to a special effect, SOUND_MAN calls that effect. When a note is finished, SOUND_MAN, using the data area's next note pointer, moves data for the next note of the sound into the area. If SOUND_MAN reads a header byte (in Cart ROM) that has bits 3 and 4 set, indicating repeat sound, it will start the sound again by reloading the first note in the sound.

After the operations upon a data area have been performed, if necessary, the channel data area pointers (PTR_TO_S_ON_x) are updated. The following data areas are processed in the same fashion, in order of occurrence, until the end of data area code, 00, is reached.

SOUND_MAN does not output the modified frequency and attenuation data. PLAY_SONGS is called just before SOUND_MAN to do this.

Special codes in byte 0 of the sound data area indicate:

255: Data area inactive, do no processing;

62: A special effect is to be played; SOUND_MAN calls the effect routine;

0: End of sound data areas (SOUND_MAN processes data areas until it sees 0 in byte 0).

NOTE: Sound number 1 MUST use the first area in the block of sound data areas. SOUND_INIT uses this address to find the sound data area.

INPUT: None.

CALLS: PT_IX_TO_SxDATA,
PROCESS_DATA_AREA.

OUTPUT: Calls routines which:
1. Decrement sound duration and sweep timers.

2. Modify swept frequency and attenuation values.
3. Call special effects routines where necessary.
4. Update the channel data area pointers if necessary.
5. Restart the sound if indicated.

7.5 PLAY_SONGS:

PLAY_SONGS takes the frequency and attenuation data pointed to by the four channel data area pointers (PTR_TO_S_ON_X) and outputs it to the four sound chip generators.

INPUT: None.

CALLS: TONE_OUT, UPATNCTRL.

OUTPUT: 1. Current frequency and attenuation data is output to

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

each tone generator, if sound
on that channel is active; if
sound on that channel is
inactive, that generator is
turned off.

2. Noise generator is sent
current attenuation data and
control data, if new.
3. Modifies SAVE_CTRL if
necessary.

7.6 Application

These four routines would normally be called as follows:

Begin

Power on inits done by OS

Cartridge program receives control

LD B, number of song data areas used in the
game

LD HL, address where LST_OF_SND_ADDRS is
store in ROM.

1	CALL SOUND_INIT to initialize sound data areas
2	Whatever other power on inits you want to do
3	Start game
4	.
5	.
6	.
7	LD B, number of sound you want to start
8	CALL PLAY_IT to set up for start of sound
9	.
10	.
11	VDP interrupt occurs:
12	CALL PLAY_SONGS to output data
13	CALL SOUND_MAN to process sound data
14	Whatever else you want to do during VDP
15	interrupt
16	RETN to game
17	End
18	
19	
20	
21	
22	
23	
24	
25	
26	

SECTION VIII
BOOT-UP SOFTWARE

8.1 Power-Up Procedure

```
begin (*run from 0*)
    set up stack_pointer
    (*power up*)
        if cartridge type = test
            execute the code at starting address
            found in location 800AH (Logo Bypass)
        else
            disable sound chip
            init random number generator
            init controller buffer areas
            defer writes = false
            mux sprites = false
            (*display_logo*)
                fill VRAM with 0's
                set up VDP to mode 1
                load ASCII generators
```

```
1      load logo generators
2      load logo names
3      load logo colors
4      enable display
5      if cartridge = game
6          display logo and game name
7          wait 12 seconds
8          disable display
9          execute the code at starting
10         address found in location 800AH
11     else (*cartridge not present*)
12         display log and "insert cartridge"
13         message
14         wait 60 seconds
15         disable display
16         soft halt
17     endif (*cartridge = game*)
18     endif (*cartridge type = test*)
19     end (*run from 0*)
20
21
22
23
24
25
26
```


8.2 Title Screen

During the power-up process, the boot-up software will look for an ASCII string of characters at Cartridge ROM location GAME_NAME for display on the logo screen.

The following information should be in the string:

1. Cartridge title with trademark (T=1EH, M=1FH).
2. Original licensor of the game.
3. The year the cartridge is released.

Example:

```
DEFB  "DONKEY KONG JUNIOR",1EH,1FH
DEFB  /PRESENTS NINTENDO'S/1983"
```

Each string is delimited by a slash (/). The first two strings are limited to 28 characters and the last string is four characters.

8.3 Cartridge Present Identifier:

All cartridges must store OAAH at location 8000H for the OS to recognize them as cartridges that require logo display.

1 The OS will initialize portions of the hardware, select
2 data areas, display the logo screen and then pass
3 control to the cartridge program.
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

SECTION IX

MISCELLANEOUS UTILITIES

9.1 ADD816

Calling Sequence:

LD A, VALUE

LD HL, ADDRESS

CALL ADD816

Description:

ADD816 adds an 8-bit signed number in accumulator to a 16-bit unsigned number pointed to by HL; returns with altered 16-bit number at the HL address.

Parameters:

VALUE 8-bit signed number.

ADDRESS

Address pointing to a 16-bit
unsigned number

Output:

Two-byte value at the address
pointed to by the HL register
pair.

Side Effects:

Destroys registers A, F and B.

9.2 DECLSN

Calling Sequence:

LD HL, ADDRESS

CALL DECLSN

Description:

DECLSN decrements least significant nibble of a byte pointed to by HL without affecting most significant nibble or HL. Returns with altered 8-bit number at HL address. Sets Z-flag if 0, C-flag if -1.

Parameters:

ADDRESS

Address pointing to an 8-bit unsigned number.

Output:

A one-byte value at the address pointed to by the HL register pair.

Side Effects:

Destroys A and F.

1

2

9.3 DECMSN

3

4

Calling Sequence:

5

6

LD HL, ADDRESS

7

CALL DECMSN

8

9

Description:

10

11

DECMSN decrements the most significant nibble of byte pointed to by HL without affecting the least significant nibble or HL. Returns with altered 8-bit number at HL address. Sets Z-flag if 0, C-flag if -1.

12

13

14

15

16

Parameters:

17

18

ADDRESS

Address pointing to 8-bit unsigned number.

19

20

21

Output:

A one-byte value at the address pointed to by the HL register pair.

22

23

24

25

Side Effects:

26

Destroys A and F.

9.4 MSNTOLSN

Calling Sequence:

LD HL, ADDRESS

CALL MSNTOLSN

Description:

MSNTOLSN copies the most significant nibble of byte pointed to by HL to the least significant nibble of that byte. The routine returns the results at the location pointed to by HL.

Parameters:

ADDRESS Address pointing to an 8-bit unsigned number.

Output: A one-byte value pointed to by HL register pair.

Side Effects:

Destroys A, F and B.

9.5 RAND_GEN

Calling Sequence:

CALL RAND_GEN

Description:

RAND_GEN is a 16-bit psuedo random number generator. It "exclusive OR's" the 15th and 8th bit together and then rotates the entire quantity to the left and inserts the "exclusive OR'ed" bit into the rightmost bit. Upon leaving, it stores the random number at global location RAND_NUM.

Output:

The random number can be found in the HL register pair or RAND_GEN because RAND_GEN contains the value of L while RAND_GEN + 1 has the value of H, or in the accumulator because A = L before RET.

Side Effects:

Destroys registers AF and HL (return values).

1 9.6 LOAD ASCII

2
3 Calling Sequence:

4
5 CALL LOAD_ASCII

6
7 Description:

8
9 LOAD_ASCII writes out the ASCII generator set to the
10 pattern generator table. The ASCII table is located in
11 Cartridge ROM starting at ASC_TABLE. INIT_TABLE must be
12 called to set up the table addresses before using this
13 routine.

14
15 Side Effects:

16 Destroys AF, DE, HL and IY.

17
18 Calls to other OS routines:

19 - PUT_VRAM
20
21
22
23
24
25
26

SECTION X

DEFINED REFERENCE LOCATIONS

10.1 OS ROM

In the OS ROM area, it is IMPORTANT to know that the application programs should only use the OS entry points listed in the OS_SYMBOLS file. Accessing to the OS otherwise is illegal and may cause program malfunction when hardware configuration changes or OS routines relocated due to update. The jump table starts from location JUMP_TABLE through the end of OS ROM. It contains all the subroutine entry points released to the user.

At the beginning of the cartridge, there are eight programmable restarts at addresses 0008H, 0010H, 0018H, 0020H, 0028H and 0030H. Each of the restarts jump to a location in Cartridge ROM where a vector can be provided to access an OS entry point. The Z80A-CPU hardware also designates location 0038H to service maskable interrupt

1 (MI) and location 0066H to service non-maskable
2 interrupt (NMI). Jump instructions are provided for
3 these two reference locations for the user to implement
4 interrupt vectors in Cartridge ROM. Starting at
5 location 0069H is the OS ROM data area which contains
6 the AMERICA byte, ASCII table address and numeric table
7 address. Figure 10-1 is the OS ROM map showing all the
8 reference locations mentioned above. Appendix E lists
9 all entry points of the Jump Table.

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

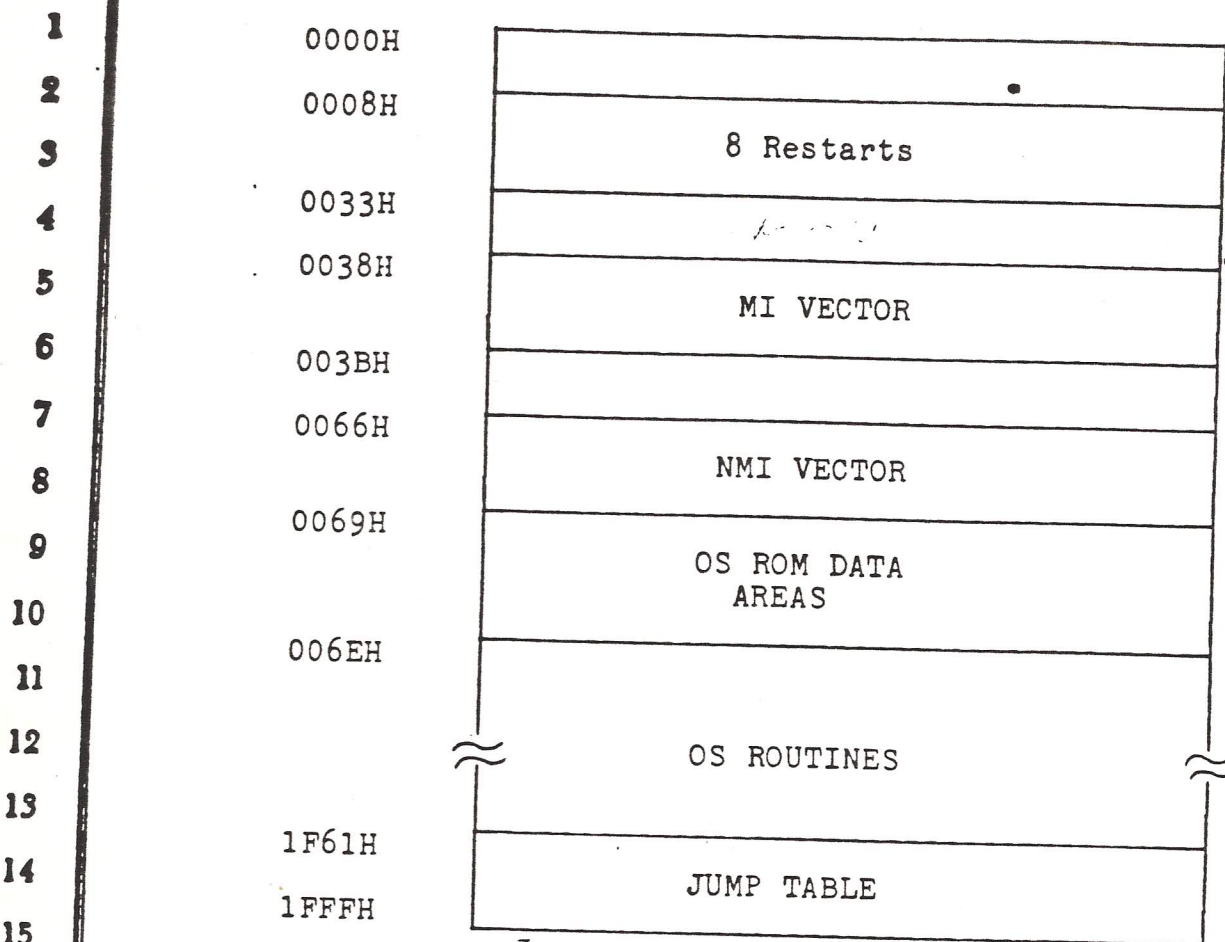


Figure 10-1

OS ROM MAP

10.1.1 Europe/America Byte:

The European TV uses PAL system (625-line format) which requires interrupt at the end of each active-display scan every 1/50 second, as opposed to every 1/60 second for the US model (NTSC, 525-line format).

1
2 ColecoVision cartridges must be interchangeable between
3 both systems, the Europe/America byte at AMERICA in OS
4 ROM, has been established to detect which version of the
5 unit is in use. If a real-time display (such as a
6 clock) must be implemented, the program will have to
7 access the Europe/America byte to determine the current
8 line frequency. For America-based units, this location
9 will contain 60 (3CH) and for European-based units, it
10 will contain 50 (32H).
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

10.1.2 Restart Vectors

Figure 10-2 shows the eight programmable restarts their addresses and corresponding locations in Cartridge ROM.

<u>OS ADDRESS</u>	<u>JUMP TO CART. ROM ADDR.</u>
0008H	800CH
0010H	800FH
0018H	8012H
0020H	8015H
0028H	8018H
0030H	801BH

Figure 10-2

OS RESTARTS

For each of the restart locations above, there should be a vector in Cartridge ROM provided by the user. To use a restart, the user must place a jump instruction to the address of the routine which he or she wishes to access through the Cartridge ROM vector; for example, JP WRITE_ VRAM at 800CH. These routines are usually the ones most frequently used in order to save application program space.

10.1.3 Graphics Tables

There are two graphics tables in the OS available to the user. The pointers for the ASCII table and Number table are defined in the locations of ASCII-TABLE and NUMBER_TABLE.

The ASCII table contains pattern generators for all 26 upper and psuedo-lower (half-size upper) case letters plus eleven special characters in 5x7 dot matrix form. The number table contains pattern generators for the numbers from 0 to 9 plus seven special characters.

10.2 Cartridge ROM

At the beginning of Cartridge ROM, locations are reserved for testing cartridge presence (Section 8-3), plus a number of pointers which point to tables, buffers and start of the game. On top of the pointers there are

spaces allocated for restart (Ref. Figure 10-2) and interrupt vectors. There are up to 60 bytes available to the user starting at location GAME_NAME, to name the cartridge, their format has been described in the title screen in section 8.2. Figure 10-3 shows the cartridge ROM map.

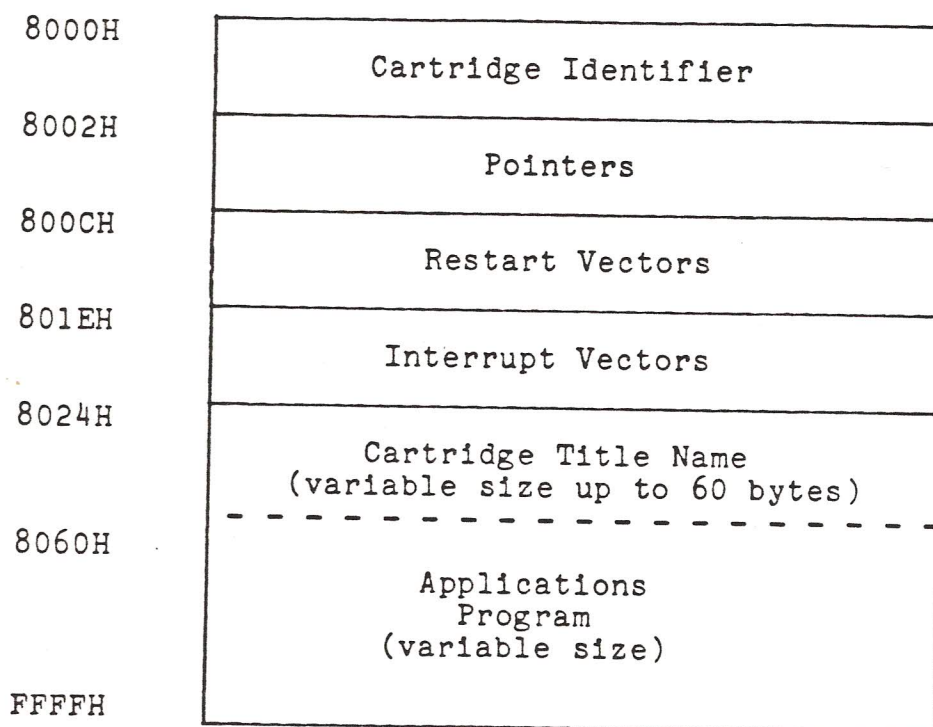


Figure 10-3

CARTRIDGE ROM MAP

10.3 CRAM Areas

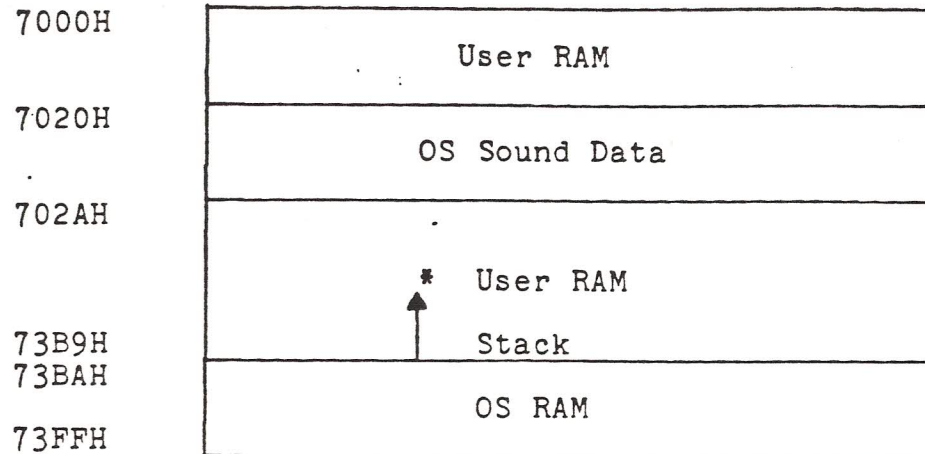


Figure 10-4

CRAM MAP

Figure 10-4 is the CRAM Map. Eleven bytes are reserved for OS sound data starting at 7020H; seventy-one bytes at the high end of memory are used by various OS routines. The top of the stack is sitting at address 73B9H which grows in the decrementing direction. Between stack and user buffer there are 942 bytes available for the application program. However, care should be exercised in both size and boundary when using CRAM as scratch pad.

Table 10-1 lists all reserved CRAM areas for user reference.

Table 10-1

DETAILED CRAM REFERENCE LOCATIONS

1		7000H	(Start of user RAM)
2			
3			
4	PTR_TO_LST_OF_SND_ADDRS	7020H	(OS Sound Data Area)
5	+1	7021H	
6	PTR_TO_S_ON_0	7022H	
7	+1	7023H	
8	PTR_TO_S_ON_1	7024H	
9	+1	7025H	
10	PTR_TO_S_ON_2	7026H	
11	+1	7027H	
12	PTR_TO_S_ON_3	7028H	
13	+1	7029H	
14	SAVE_CTRL	702AH	
15		702BH	(Resume user RAM)
16	STACK	73B9H	(Top of Stack)
17	PARAM_AREA	73BAH	(Parameter passing area for
18	+1	73BBH	Pascal calls to OS routines)
19	+2	73BCH	
20	+3	73BDH	

21

22

23

24

25

26

1	+4	73BEH
2	+5	73BFH
3	+6	73COH
4	+7	73C1H
5	+8	73C2H
6	VDP_MODE_WORD	73C3H
7	+1	73C4H
8	VDP_STATUS BYTE	73C5H
9	DEFER_WRITES	73C6H
10	MUX_SPRITES	73C7H
11	RAND_NUM	73C8H
12	+1	73C9H
13	QUEUE_SIZE	73CAH
14	QUEUE_HEAD	73CBH
15	QUEUE_TAIL	73CCH
16	HEAD_ADDRESS	73CDH
17	+1	73CEH
18	TAIL_ADDRESS	73CFH
19	+1	73DOH
20	BUFFER	73D1H
21	+1	73D2H
22	TIMER_TABLE_BAS	73D3H
23		
24		
25		
26		

1	+1	73D4H
2	NEXT_TIMER_DATA	73D5H
3	+1	73D6H
4	DBNCE_BUFF	73D7H (FIRE_OLD - Player 0)
5	+1	73D8H (FIRE_STATE - Player 0)
6	+2	73D9H(JOY_OLD - Player 0)
7	+3	73DAH (JOY_STATE - Player 0)
8	+4	73DBH(SPIN_OLD - Player 0)
9	+5	73DCH(SPIN_STATE - Player 0)
10	+6	73DDH(ARM_OLD - Player 0)
11	+7	73DEH(ARM_STATE - Player 0)
12	+8	73DFH(KBD_OLD - Player 0)
13	+9	73E0H(KBD_STATE - Player 0)
14	+10	73E1H(FIRE_OLD - Player 1)
15	+11	73E2H(FIRE_STATE - Player 1)
16	+12	73E3H(JOY_OLD - Player 1)
17	+13	73E4H(JOY_STATE - Player 1)
18	+14	73E5H(SPIN_OLD - Player 1)
19	+15	73E6H(SPIN_STATE - Player 1)
20	+16	73E7H(ARM_OLD - Player 1)
21	+17	73E8H(ARM_STATE - Player 1)
22	+18	73E9H(KBD_OLD - Player 1)
23		
24		
25		
26		

1	+19	73EAH	(KBD_STATE - Player 1)
2	SPIN_SWO_CT	73EBH	.
3	SPIN_SW1_CT	73ECH	
4	STROBE_FLG	73EDH	
5	SO_CO	73EEH	
6	SO_C1	73EFH	
7	S1_CO	73F0H	
8	S1_C1	73F1H	
9	VRAM_ADDR_TABLE	73F2H	
10	SPRITENAMETBL	73F2H	
11	+1	73F3H	
12	SPRITEGENTBL	73F4H	
13	+1	73F5H	
14	PATTRNNAMETBL	73F6H	
15	+1	73F7H	
16	PATTRNGENTBL	73F8H	
17	+1	73F9H	
18	COLORTABLE	73FAH	
19	+1	73FBH	
20	SAVE_TEMP	73FCH	
21	+1	73FDH	
22	SAVED_COUNT	73FEH	
23	+1	73FFH	
24			
25			
26			

. APPENDIX B

COLECOVISION

GRAPHICS USERS' MANUAL

11/30/82

V1.0

COLICOVISION
GRAPHICS USERS' MANUAL

Table of contents

<u>Page</u>	<u>Section</u>
1	1.0 Introduction
1	1.1 Summary of Object Types
2	1.2 Data structure Overview
2	1.3 Graphics Routines Overview
3	1.4 Some Key Concepts
discussion of object types	
5	2.0 Semi-Mobile Objects
6	2.1 Detailed Description of Semi-Mobile Object Data Structure
12	3.0 Mobile Objects
13	3.1 Detailed Description of Mobile Object Data Structure
17	4.0 Sprite Objects
18	4.1 Detailed Description of Sprite Object Data Structure
21	5.0 Complex Objects
21	5.1 Detailed Description of Complex Object Data Structure
discussion of graphics routines	
24	6.0 Activate
24	6.1 Description of Activate Routine
28	7.0 Put_Object
28	7.1 Description of Put_Object
28	7.2 Description of Put_Semi
29	7.3 Put_Mobile
30	7.4 Description of Put_Mobile
34	7.5 Put_Sprite
34	7.6 Put_Complex
data structure summary	
36	8.0 Data Structure Summary for Semi-Mobile Objects
39	9.0 Data Structure Summary for Mobile Objects
41	10.0 Data Structure Summary for Sprite Objects
42	11.0 Data Structure Summary for Complex Objects
figures	
44	figure 1
45	figure 2
46	figures 3, 4, 5

1.0 INTRODUCTION

The material in this manual assumes that the reader is familiar with Texas Instruments' 9918/9928 Video Display Processor (VDP). Since the system routines assume that the VDP will be operating in Graphics Mode I or II, particular attention should be given to the discussion of these modes.

The Colecovision operating system includes several graphics routines to help facilitate the creation and manipulation of images on the display. These routines and associated data structures enable the cartridge programmer to treat graphic elements as conceptual entities called "objects", of which there are four types. Each object may consist of one or more "frames". A frame is a single, visual manifestation of the object.

The graphics routines provide a high-level means of altering the frame displayed and the position of objects.

1.1 SUMMARY OF OBJECT TYPES

The four possible object types are:

1. Semi-Mobile

Semi-Mobile objects are rectangular arrays of pattern blocks. They are always aligned on pattern boundaries but may bleed on and off the pattern plane in any direction. Semi-Mobile objects may be used either for moving images (when incremental motion by pattern plane positions is acceptable) or for setting up background patterns.

2. Mobile

The primary purpose of Mobile objects is to overcome a particular limitation of the VDP which prevents more than 4 sprites from being displayed on a given horizontal TV line. Mobile objects are always 2 by 2 pattern blocks in size. They may be placed anywhere on the pattern plane with pixel resolution and will appear as superimposed upon the background. They also may bleed on and off the pattern plane in any direction.

3. Sprite

Sprite objects are composed of individual sprites.

4. Complex

Complex objects are collections of other "component" objects. The component objects may be of any type including other Complex objects.

1.2 DATA STRUCTURE OVERVIEW

Each object is defined by a "high level definition" in cartridge ROM (CROM) which links together several different data areas. The data contained within these areas completely specifies all aspects of an object. The following is a brief description of the different areas.

GRAPHICS

This data area is also located in CROM. Pattern and color generators for Semi-Mobile, Mobile and Sprite objects, and frame data for all objects are located in the GRAPHICS data area. The data structure within each GRAPHICS area depends on the type of object with which it is associated. If, however, two or more objects of the same type are graphically identical, they may share the same GRAPHICS data area. This will reduce the amount of graphics data that needs to be stored in CROM.

STATUS

Each object must have its own STATUS area in CPU RAM. The cartridge program uses this area to manipulate the object. This is done by altering the location within STATUS which determines which frame is to be displayed as well as the locations which define the position of the object on the display. The graphics routine, PUT_OBJECT, when called, will access the object's status area and place the object accordingly.

OLD_SCREEN

Mobile and Semi-Mobile objects utilize the pattern plane. They are displayed by overwriting an array of the names in the pattern name table with an array of names which represents a particular frame of the object. The overwritten names can be thought of as representing a background frame which is "underneath" the object. If the background frame will need to be restored to the display (e.g. when the object moves or is removed from the display) then it is necessary to save that frame. OLD_SCREEN is a save area for background frames. The graphics routines PUT_SEMI and PUT_MOBILE will automatically save and restore background frames when an object is moved or its frame is changed. OLD_SCREEN may be located in CROM or in VRAM.

1.3 GRAPHICS ROUTINES OVERVIEW

ACTIVATE

The primary purpose of this routine is to move the pattern and color generators from the GRAPHICS data area into the pattern and color generator tables in VRAM. Each object must be "activated" before it can be displayed. ACTIVATE also initializes the first byte in an object's OLD_SCREEN data area with the value 80H. PUT_OBJECT tests this location before restoring the background names to the name table. If the value 80H is found, it is an indication that the object has not yet been displayed and therefore, there are no saved background names in OLD_SCREEN. ACTIVATE initializes a byte (in the case of Mobile objects, a word) which indicates where additional generators should be

located if such generators are to be created at game-on time by other routines, such as REFLECT_VERTICAL (described elsewhere in the COLECOVISION USERS MANUAL). Finally, ACTIVATE will initialize the FRAME variable in the object's STATUS area to 0.

PUT_OBJECT

This routine is called when an object's frame or its location on the display is to be changed. The routine tests the type of object and then branches to one of four other routines designed to handle that particular object type. These routines function as follows:

1. PUT_SEMI

Semi-Mobile objects are placed on the display by writing the generator names specified by one of the object's frames into the pattern name table in VRAM. The pattern and color generators which are needed to create the frame must already be in their respective generator tables.

2. PUT_MOBILE

Mobile objects are displayed by producing a new set of pattern and color generators which depict the frame to be displayed on the background. These new generators are then moved to the locations in the VRAM pattern and color generator tables which are reserved for the object; the names of the new generators are then written into the pattern name table.

3. PUT_SPRITE0

4. PUT_SPRITE1

These routines handle the display of size 0 and size 1 Sprite objects.

5. PUT_COMPLEX

Complex objects are aggregates of other "component" objects. The positional relationship of these component objects is defined in an offset list. For each of the component objects, PUT_COMPLEX calculates the correct X and Y location, then calls PUT_OBJECT with the address of the high-level definition for that component object passed in the IX register.

1.4 SOME KEY CONCEPTS

META-PLANE

In order to facilitate moving objects on and off the pattern plane, a conceptually larger "meta-plane" has been implemented. Positions on the meta-plane are defined with respect to two orthogonal axes, X and Y. The pattern plane is contained within the meta-plane and its origin is coincident with the origin of the meta-plane (see fig. 1).

X_LOCATION

Y_LOCATION

These two variables are part of each object's STATUS area. Each variable contains a 16 bit signed number which represents the distance

in pixels from the origin of the meta-plane. The two variables together form the coordinate location of the object's upper left corner on the meta-plane. Sprite and Mobile objects will be displayed at the exact location indicated by their X and Y_LOCATIONS. However, since Semi-Mobile objects are always aligned on pattern position boundaries, they will be displayed aligned with the nearest pattern boundary above and to the left of the indicated X and Y_LOCATIONS. When a Complex object is to be displayed, the X and Y_LOCATION of each of its component objects is computed by adding a displacement for that component to the Complex object's X and Y_LOCATIONS. Each component object is then displayed at the computed location.

To move an object, the X and or Y_LOCATIONS in the CRAM STATUS area are changed and then PUT_OBJECT is called with the address of the object's high-level definition passed in the IX register. When moving Mobile objects an additional parameter is passed in register B. This is discussed further in the description of PUT_MOBILE.

FRAME

This variable is also part of each object's STATUS area. The value contained in FRAME is used by PUT_OBJECT to select one of several pointers (or, in the case of Complex objects, pairs of pointers) which point to the data defining the graphic content of the frame. The pointers may either point to frame data which is part of the original ROMed GRAPICS, or they may point to an area in CPU RAM. In the latter case, the frame data must be created by the cartridge program before that frame can be displayed.

The frame of an object is changed by altering the FRAME variable in the object's STATUS area. PUT_OBJECT is then called in the same manner as when moving an object. When changing the frame number of a Mobile object, bit 7 of FRAME should not be altered. This bit is reserved for use by the PUT_MOBILE routine.

Since Semi-Mobile objects are displayed by altering names in the pattern name table, it may be necessary to save the pattern plane graphic which is lost in the process of displaying the Semi-Mobile object (see previous discussion of OLD_SCREEN). The third address in the object's high level definition designates a location for saving the overwritten names. If that address is greater than or equal to 7000H, then the OLD_SCREEN will reside in CPU RAM. If the address is less than 7000H, then OLD_SCREEN will be in VRAM. Finally, if the address is 8000H or greater, then the overwritten names will not be saved.

2.1 DETAILED DESCRIPTION OF SEMI-MOBILE OBJECT DATA STRUCTURE

(Identifiers in caps refer to symbols in the data structure summary)

Each Semi-Mobile object is defined in cartridge ROM by:

- 1) SMO - the object's "high level definition"
- 2) GRAPHICS - an area containing the object's graphics data, divided into three subsections:
 - Parameters and Pointers
 - Frames
 - Generators

and in CPU RAM by:

- 1) STATUS - a status area
- 2) OLD_SCREEN - an optional location for saving overwritten backgrounds. (OLD_SCREEN may alternatively be stored in VRAM)

A detailed description of each structure follows.

*** SMO - cartridge ROM

The high level definition, at SMO, is composed of three 16 bit addresses; these are stored in the normal manner with the low order byte first:

byte:

- | | |
|---|--|
| 0 | address of GRAPHICS (the start of the ROMed graphics data for the object) |
| 2 | address of STATUS (the object's RAM status area) |
| 4 | address of OLD_SCREEN (VRAM or CPU RAM area for saving background information; bit 15 of the OLD_SCREEN address is a flag indicating whether or not backgrounds are to be saved; if bit 15 is set, backgrounds will not be saved). |

*** GRAPHICS - cartridge ROM

The ROMed graphics data for a Semi-Mobile object can be thought of conceptually as three chunks. Each chunk is stored as follows:

*** Parameters and Pointers

byte:

- Parameters -

- 0 OBJ_TYPE - OBJ_TYPE is divided into two parts:
LSN - specifies the object type which, for Semi-Mobile objects, must equal 0.
MSN - only meaningful when the VDP is operating in graphics mode II. Bits 5, 6 and 7 indicate which third or thirds of the pattern plane the object (or any part of the object) may be required to appear. This information is used by routines which move the object's pattern and color generator data to VRAM.
bit 7 - if set, generators will be moved to the 1st third of the generator tables.
bit 6 - if set, generators will be moved to the 2nd third.
bit 5 - if set, generators will be moved to the 3rd third.
bit 4 - indicates the number of color generator bytes per 8 byte pattern generator which are included as part of the ROM graphics data. If this bit is 0, then there must be 8 color generator bytes per pattern generator (the normal case for graphics mode II). If bit 4 is 1, then only 1 color generator byte per pattern generator will be expected, giving the programmer the option of reducing the number of ROMed color generator bytes needed when operating in graphics mode II. The single color byte will be expanded to 8 bytes by the routine which moves the color generators to VRAM.
- 1 FIRST_GEN_NAME - an index from the start of the pattern and color generator tables in VRAM. This index specifies the location to which the ROMed pattern and color generators will be moved (i.e. the base address of the pattern generator table + $8 * \text{FIRST\_GEN\_NAME}$ = the address within the pattern generator table where the object's 1st pattern generator will be stored. This is also true for the color generator table. The first pattern generator will be moved to the location in the pattern generator table indexed by FIRST_GEN_NAME and the rest of the generators will be loaded sequentially. The color generators are loaded in a similar fashion.
- 2 NUMGEN - indicates how many pattern/color generator pairs are defined (stored) in the graphics data area. This is equal to the number of generator pairs which will be moved to the VRAM generator tables.
- Pointers -
- 3 address of GENERATORS - the start of the ROMed pattern and color generators in the object's graphics data area. Color generators must be stored immediately after pattern generators; therefore, the address of the first color generator within the graphics data area can be computed from NUMGEN and GENERATORS.

- 5 address of FRAME_0 - FRAME_0 is the address at which the data specifying the object's first frame is stored. As indicated by the frame address, the data for a given frame may be stored in ROM (as part of the graphics data area) or it may be stored in CPU RAM. RAM storage of frame data allows for the algorithmic generation of additional frames at game on time: e.g., rather than storing both a frame and its rotated version in ROM, ROM space can be saved by storing only one frame and generating the rotated frame data in RAM. Also, frame data stored in RAM can be dynamically modified, allowing for special frame effects (e.g. color modification).
- 7 address of FRAME_1 - the address at which the data specifying the object's second frame is stored.
- :
- :
- 2n+5 address of FRAME_n - the object's last frame

*** Frames - cartridge ROM or CPU RAM

Since the frame pointers (FRAME_0...FRAME_n) are 16 bit addresses, the frame data for an object may be located anywhere in cartridge ROM or RAM. The format of a frame's data is as follows:

byte:

- 0 X_EXTENT - specifies the width of the frame in pattern plane positions, its "X_EXTENT", which must be > 0 and ≤ 255 . As indicated by the range of values for the X_EXTENT, a frame may be much greater in width than can be displayed within the pattern plane. When a frame with a X_EXTENT which is greater than 32 is to be displayed, the section actually visible will depend on the specified X position (i.e. if a frame of an object has a X_EXTENT of 64 and the X position of the object is -8×32 pixels, then the right half of the frame will be displayed on the pattern plane). This feature allows objects to be scrolled horizontally by creating a frame greater in width than the pattern plane and then displaying the object with varying values of the X position.
- 1 Y_EXTENT - specifies the height of the frame in pattern plane positions, its "Y_EXTENT", which must be > 0 and ≤ 255 . As indicated by the range of values for the Y_EXTENT, a frame may be much greater in height than can be displayed within the pattern plane. When a frame with a Y_EXTENT which is greater than 24 is to be displayed, the section actually visible will depend on the specified Y position (i.e. if a frame of an object has a Y_EXTENT of 48 and the Y position of the object is -8×24 pixels, then the bottom half of the frame will be

displayed on the pattern plane). This feature allows objects to be scrolled vertically by creating a frame greater in height than the pattern plane and then displaying the object with varying values of the Y position.

2... for X_EXTENT * Y_EXTENT bytes

Following the X and Y_EXTENT is an array of the pattern names (length in bytes = X_EXTENT * Y_EXTENT) which specify the generators used to create that frame. The names are arranged in row major order (i.e. the first X_EXTENT names are the names of the generators which will be displayed in the first row of the frame, the second X_EXTENT names are the names of the generators which will be displayed in the second row of the frame, etc.). There must be exactly X_EXTENT by Y_EXTENT names in the array.

For Semi-Mobile objects the names stored in the above described name list are "names" in the TI sense of the word: i.e., they are values ranging from 0 to 255 that directly point to (or "name") a generator location within the pattern generator table.

*** Generators - cartridge ROM

All the ROMed pattern and color generators must be grouped together in a contiguous block (starting at location GENERATORS). Each pattern generator must be 8 bytes long, and the number of pattern generators must conform to the value stored in NUMGEN:

byte:

0 bb,bb,bb,bb,bb,bb,bb,bb - the first 8 byte pattern generator (bb = binary graphic data)
 8 bb,bb,bb,bb,bb,bb,bb,bb - the second 8 byte pattern generator
 etc. for a total of 8*NUMGEN bytes

The color generators are stored immediately following the pattern generators. The format of the color generators depends on which graphics mode is being used and whether bit 4 of OBJ_TYPE is set or not.

GRAPHICS MODE II color generator storage:

When OBJ_TYPE bit 4 = 0, there must be eight color generator bytes per pattern generator.

byte:

0-7 bb,bb,bb,bb,bb,bb,bb,bb - color generator bytes for 1st pattern
 8-15 bb,bb,bb,bb,bb,bb,bb,bb - color generator bytes for 2nd pattern
 etc., for a total of 8*NUMGEN bytes

When OBJ_TYPE bit 4 = 1, there must be only 1 color generator byte per pattern generator. It will be expanded to 8 bytes when moved to the VRAM color generator table. This feature is useful if each generator for a particular object can use the same two color combination for all 8 lines.

Byte:

0 bb - color generator byte for 1st pattern
1 bb - color generator byte for 2nd pattern
etc., for a total of NUMGEN bytes.

GRAPHICS MODE I color generator storage:

In Graphics Mode I the pattern generator table is divided into 32 "groups" of 8 (contiguous) generators (see TI VDP manual page 18). All the generators within a group share the same color generator byte. Therefore, there must be one color generator byte stored per group occupied by the object's pattern generators. The first color generator byte will be placed in the color generator table at an offset of FIRST_GEN_NAME/8 and the rest will be placed in sequential locations.

NOTE 1: The exact number of color generator bytes needed by an object depends both on the number of pattern generators contained in the graphics data and the location in the pattern generator table to which the generators will be moved. For example, if an object has 8 pattern generators and they are moved to the pattern generator table starting at location 0 (offset from the start of the table), then only 1 color generator is needed. However, if the same 8 pattern generators are loaded into the pattern generator table starting at location 20H, then 2 color generators will be needed, since the first four generators are in one group and the second four are in another group.

NOTE 2: Due to an error in the ACTIVATE routine, ACTIVATE cannot be used to move pattern and color generators of Semi-Mobile objects to VRAM when the VDP is operating in graphics mode I, and when FIRST_GEN_NAME is equal to or greater than 80H. In this situation the cartridge program must fulfill the functions of ACTIVATE (see discussion of ACTIVATE).

*** STATUS - CPU RAM

An object's status area consists of 4 elements:

Byte:

0 1 byte for FRAME - indicates which of the object's frames is to be displayed.
1 2 bytes for X_LOCATION

3 2 bytes for Y_LOCATION - X_LOCATION and Y_LOCATION give the coordinate position of the upper left corner of the object on a "metaplane" which includes the pattern plane (see fig. 1). The origin of the pattern plane is coincident with the origin of the metaplane. The values at X_LOCATION and Y_LOCATION are 16 bit signed numbers which permit the positioning of an object anywhere within, or outside of, the pattern plane. This enables an object to be bled on or off the pattern plane in any direction.

5 1 byte for NEXT_GEN - an index which points to the next generator location which can be used when adding new \ generators to an object's generator tables. After the object's ROMed generators have been moved to its VRAM tables, ACTIVATE will set NEXT_GEN equal to FIRST_GEN_NAME + NUMGEN.

Some objects may require generators which are essentially modified versions of other generators (e.g. a generator which represents the pattern of another generator which has been rotated). ROM space can be conserved by including only the "unmodified" generators in the graphics data and using system routines to generate the modified versions. NEXT_GEN points to the next free location in that object's generator table to which a modified generator may be added. When adding new generators in this manner, NEXT_GEN should be updated in order to prevent new generators overwriting old ones.

*** OLD_SCREEN - VRAM or CPU RAM

OLD_SCREEN is a buffer for saving the pattern names which are overwritten in the process of displaying an object. These names constitute a "background frame" which has the same X and Y_EXTENTS as the frame of the object which is currently being displayed. The data structure within OLD_SCREEN is the same as the data structure for a frame with two extra bytes tacked on at the beginning. These bytes, X_PAT_POS and Y_PAT_POS, indicate where on the pattern plane this background frame belongs, and are expressed in terms of pattern plane positions. The next time the position or frame of this object is changed PUT_OBJECT will restore the background frame to the display and save a "new" background frame before placing the object on the pattern plane.

byte:

0 1 byte for X_PAT_POS - the column in which the upper left corner of the saved background screen (frame) lies
1 1 byte for Y_PAT_POS - the row in which the upper left corner of the saved background screen (frame) lies
2 1 byte for X_EXTENT of the saved screen - set by PUT_OBJECT to equal X_EXTENT of the frame which eclipses it
3 1 byte for Y_EXTENT of the saved screen - set by PUT_OBJECT to equal Y_EXTENT of the frame which eclipses it
4 n bytes for storage of pattern names; where n = the number of patterns contained in the largest frame of the object (i.e., $n = X\_EXTENT * Y\_EXTENT$ for the object's largest frame)

3.0 MOBILE OBJECTS

Mobile objects emulate size 1 sprites with 0 magnification (i.e. they are always 16 by 16 pixels in size and appear as if they were superimposed upon the background). They may be positioned anywhere on the pattern plane with pixel resolution and may also be made to bleed on and off the pattern plane in any direction.

Each frame of a Mobile object requires exactly four⁰ pattern generators and one color generator. These generators, however, are not moved to VRAM. In order to display the object anywhere with pixel resolution, a new set of nine pattern and color generators must be created by the OS graphics routine PUT_MOBILE. These new generators represent graphically the superposition of the Mobile object upon the background at which it is to be displayed. The new generators are then moved to the VRAM pattern and color generator tables, and next the names of these generators are written into the pattern name table, thus displaying the object at the desired location. Nine generators are used in order to cover all positional relationships between the desired position of the object and the boundaries of the pattern positions in the pattern plane (see figure 1).

The pattern and color generator table space used by PUT_MOBILE depends on the graphics mode being used and, in graphics mode II, the location of the object on the display.

In graphics mode I, PUT_MOBILE will use 18 pattern generator locations. The first pattern generator will be located at FIRST_GEN_NAME (i.e. the actual VRAM address will be the pattern generator base address + FIRST_GEN_NAME * 8). Three or four color generator bytes will be required depending on the value of FIRST_GEN_NAME. If FIRST_GEN_NAME MOD 8 < 7, then there needs to be space for three color generators: if FIRST_GEN_NAME MOD 8 = 7, then there needs to be space for four color generators. The first of the color generators will be located at FIRST_GEN_NAME/8 (i.e. VRAM address equals the color table base address + FIRST_GEN_NAME/8).

When operating in graphics mode II, Mobile objects will require generator space for 18 pattern and 18 color generators in each third of the pattern and color generator tables which corresponds to that third of the pattern plane in which any part of the object may appear. The location within each third of the tables is determined by FIRST_GEN_NAME. When any part of the object is in the top third of the pattern plane, pattern generators will be located at the VRAM address given by the pattern generator base address + FIRST_GEN_NAME * 8. If any part is in the middle third of the pattern plane, pattern generators will be located at the VRAM address given by the pattern generator base address + 800H + FIRST_GEN_NAME * 8, and if any part is in the bottom third, pattern generators will be located at the VRAM address given by the pattern generator base address + 1000H + FIRST_GEN_NAME * 8. The color generators are located in a similar manner.

Even though only 9 pattern and 9 color generators (2 color generator bytes in graphics mode I) are "active" (being displayed) at a given time, the additional generator space is required to enable PUT_MOBILE to move new sets of pattern and color generators to VRAM without disturbing the display. While one set of 9 pattern and color generators are being displayed, the other set of 9 can be changed. After the change is completed, the new generators are displayed by writing the new pattern names into the pattern name table.

Each Mobile object requires a STATUS area in CPU RAM which contains the frame of the object to be displayed, its location on the display, and a pointer to the area for adding new generators (these new generators are used in the same manner as the object's ROMed generators). Each object also requires an OLD_SCREEN area which serves the same purpose as OLD_SCREEN areas for Semi-Mobile objects.

Even though a Mobile object's generators are not moved directly to VRAM, each object must be "activated" in order to initialize the OLD_SCREEN and STATUS areas.

To place a Mobile object on the display, the desired location should be loaded into the X and Y_LOCATION variables in its STATUS area. In addition, the FRAME variable must contain the desired frame number. When loading the frame number, however, only bits 0-6 should be used. Bit 7 should not be altered. This bit is used by PUT_MOBILE (see description of PUT_MOBILE).

PUT_OBJECT is then called, passing the address of the high-level definition in the IX register. In addition, register B is used to pass two parameters which determine the method for combining the background graphics with that of the object (see description of PUT_MOBILE).

3.1 DETAILED DESCRIPTION OF MOBILE OBJECT DATA STRUCTURE

(Identifiers in caps refer to symbols in the data structure summary)

Each Mobile object is defined in cartridge ROM by:

- 1) MOB - the object's "high level definition"
- 2) GRAPHICS - an area containing the object's graphic data, divided into three subsections:
 - Parameters and pointers
 - Frames
 - Generators

and in CPU RAM by:

- 1) STATUS - a status area
- 2) OLD_SCREEN - a location for saving overwritten background names (OLD_SCREEN may be either in CPU RAM or VRAM).

A detailed description of each structure follows.

*** MOB - cartridge ROM

The high level definition at MOB is composed of four 16 bit addresses stored in the normal manner with the low order byte first:

byte:

- 0 address of GRAPHICS (the start of the ROMed graphics data for the object). Any number of Mobile objects may use the same graphics data. However, each Mobile object must have its own STATUS and OLD_SCREEN areas.
- 2 address of STATUS (the object's RAM status area)
- 4 address of OLD_SCREEN (VRAM or CPU RAM area for saving background information)
- 6 FIRST_GEN_NAME (index to the start of the object's generator tables within the VRAM pattern and color generator tables)

*** GRAPHICS - cartridge ROM

The ROMed graphics for Mobile objects may be thought of as three separate segments. The data in each segment is defined as follows:

*** Parameters and Pointers

byte:

- Parameters -

- 0 OBJ_TYPE - for Mobile objects OBJ_TYPE = 1
- 1 NUMGEN - indicates how many pattern generators are contained within the graphics data area.
- 2 address of NEW_GEN (an area for storing new generators created at game-on time)
- 4 address of GENERATORS - the start of the ROMed pattern generators for the object

- Pointers -

- 6 address of FRAME_0 - this is the address of the start of the data for the object's first frame. This data, as well as the data for any of the object's frames, may be located anywhere in cartridge ROM or in CPU RAM. Frame data stored in RAM allows for the creation of new frames at game-on time and therefore reduces the amount of data which needs to be stored as part of the object's ROMed frame data.
- 8 address of FRAME_1 - address of the data for the object's second frame.

- 2n+6 address of FRAME_n - the object's frame.

*** Frames - each frame may be either in cartridge ROM or CPU RAM

FRAME_0...FRAME_n - cartridge ROM or CPU RAM

Since the frame pointers (FRAME_0...FRAME_n) are 16 bit addresses, the frame data for an object may be located anywhere in cartridge ROM or CPU RAM. The format for a Mobile object's frame data is as follows:

Byte:

0 list of 4 pattern names. The four names specify which of the object's patterns are to be displayed in each of the object's four quadrants as follows:

name	quadrant
1	upper left
2	lower left
3	upper right
4	lower right

The value of the name specifies the object's generators as follows:

0 = first ROMed pattern
1 = second ROMed pattern

:

NUMGEN-1 = last ROMed pattern

Name values greater than or equal to NUMGEN refer to patterns stored in the location starting at NEW_GEN in CPU RAM. A value of NUMGEN refers to the first pattern in that area. A value of NUMGEN + 1 refers to the second pattern etc.

4 The MSN of this byte determines the color of the object (i.e. the color1 of the object) and the LSN must be 0.

*** Generators - cartridge ROM

All of a Mobile object's pattern generators must be grouped together in a contiguous block starting at location GENERATORS. Each pattern generator must be 8 bytes long. The number of generators must equal the number stored in NUMGEN:

Byte:

0 bb,bb,bb,bb,bb,bb,bb,bb - first generator (bb = binary graphic data)

8 bb,bb,bb,bb,bb,bb,bb,bb - second generator
for a total of 8*NUMGEN bytes

*** STATUS - CPU RAM

A Mobile object's status area consists of 4 elements

byte:

- 0 FRAME - bits 0-6 indicate which frame is to be displayed. Bit 7 is used by the PUT_MOBILE routine and should not be altered when changing the frame number.
- 1 X_LOCATION
- 3 Y_LOCATION - The X and Y_LOCATION variables specify the coordinate position of the upper left corner of the object on a "metaplane" which includes the pattern plane.
- 5 pointer to NEW_GEN area - This pointer points to the next available space for adding new generators. It will be initialized by ACTIVATE with the 16 bit value NEW_GEN from the parameter segment of the object's GRAPHICS area. Routines which add generators to this area should increment the pointer by 8 each time a new generator is added if subsequent generators are not to overwrite previous ones.

*** OLD_SCREEN - VRAM or CPU RAM

OLD_SCREEN is an area for saving pattern names which are overwritten in the process of displaying the object. Since all Mobile objects are displayed by writing 9 names into the patten name table (except in cases in which part of the object is off the pattern plane) the size of the OLD_SCREEN area for any Mobile object is always 11 bytes. The first two bytes specify where (in pattern plane positions) the names came from and the next 9 bytes contain the 9 saved names. ACTIVATE initializes the first byte to 80H which indicates to PUT_MOBILE that no names have yet been saved.

byte:

- 0 X_PAT_POS - pattern position column in which the upper left corner of the saved "background frame" lies
- 1 Y_PAT_POS - pattern position row in which the upper left corner lies
- 2-11 the nine background names

4.0 SPRITE OBJECTS

Sprite objects are composed of individual sprites. They may be either size0 or size1 sprites, and may or may not be magnified.

The data areas that make up sprite objects are similar to those for the other object types. The high level definition contains two addresses which point to the object's GRAPHICS data and the STATUS area respectively. Following these addresses there is a byte which determines which of the 32 VDP sprites will be used to implement this object.

Each Sprite object must include a set of pattern generators which will be used to create an image on the display. These generators may be stored as part of the object's GRAPHICS data in ROM and moved to the sprite generator table in VRAM. The location within the sprite generator table to which the generators should be moved is determined by FIRST_GEN_NAME, a byte within the GRAPHICS data area (i.e. the first generator should be located at VRAM address = sprite generator base address + FIRST_GEN_NAME * 8).

Frames for a Sprite object are defined in the FRAME_TABLE. The FRAME_TABLE contains a pair of bytes for each frame of the object. The first byte specifies the color that the frame will be and the second byte determines which generator (or generators in the case of size1 sprites) will be used to define the shape.

Once the pattern generators have been moved to VRAM, a Sprite object may be displayed by setting up it's STATUS area and then calling PUT_OBJECT. To set up the STATUS area, the following must be done:

1. Set the first byte, FRAME, to the desired frame number to be displayed. This number is used to pick one of the pairs of bytes in the FRAME_TABLE which determines the color and shape to be displayed.
2. The 16 bit signed values at X_LOCATION and Y_LOCATION must be set to the desired x and y pixel positions of the upper left corner of the object.

To place the Sprite object on the screen, the following calling sequence is used:

```
LD IX,OBJECT_NAME  
CALL PUT_OBJECT
```

Where OBJECT_NAME is the address of the high level definition for the object.

4.1 DETAILED DESCRIPTION OF SPRITE OBJECT DATA STRUCTURE

(Identifiers in caps refer to symbols in the data structure summary)

Each Sprite object is defined in cartridge ROM by:

- 1) SPROBJ - the object's "high level definition"
- 2) GRAPHICS - an area containing the object's graphics data, divided into three subsections:
 - Parameters and Pointers
 - Frame_Table
 - Generators

and in CPU RAM by:

- 1) STATUS - a status area

A detailed description of each structure follows.

*** SPROBJ cartridge ROM

The high level definition, at SPROBJ, is composed of two 16 bit addresses stored in the normal manner with the low order byte first. Following the addresses is a byte which indicates which actual sprite number is used to implement the object.

byte:

- | | |
|---|--|
| 0 | address of GRAPHICS (the start of the ROMed graphics data for the object) |
| 2 | address of STATUS (the object's RAM status area) |
| 4 | SPRITE_INDEX (determines the sprite to be used for this object, i.e. 0-31) |

*** GRAPHICS - cartridge ROM

The ROMed graphics data for a Sprite object can be thought of in three conceptual chunks. Each chunk is stored as follows:

*** Parameters and Pointers

byte:

- | | |
|---|--|
| 0 | OBJ_TYPE - OBJ_TYPE is equal to 3 for all Sprite objects. |
| 1 | FIRST_GEN_NAME - an index into the sprite pattern generator table in VRAM. This index specifies the location to which the ROMed pattern generators will be moved (i.e. the base address of the sprite pattern generator table + 8 * FIRST_GEN_NAME = the address within the pattern generator table where the object's 1st pattern generator will be stored). The first pattern generator will be moved to the location in the pattern |

generator table indexed by FIRST_GEN_NAME and the rest of the generators will be loaded sequentially. When using sized sprites, FIRST_GEN_NAME must be a multiple of 4.

- 2 address of GENERATORS - the start of the ROMed pattern generators in the object's graphics data area.
- 4 NUMGEN - indicates how many pattern generators are defined (stored) in the graphics data area. This is the number of generators which will be moved to the VRAM generator table.
- 5 address of FRAME_TABLE - This is a pointer to the table containing shape and color information for each frame of the object.

*** FRAME_TABLE - cartridge ROM or CPU RAM

The FRAME_TABLE contains a pair of bytes for each frame. The first byte of each pair determines the color and the second byte points to the generator (or set of four generators in the case of sized sprites) to be used for that frame.

byte:

- 0 COLOR for frame 0
- 1 SHAPE - index to generator(s) for frame 0, e.g. the VRAM address of the generator(s) for this frame = sprite generator base address + 8 * (FIRST_GEN_NAME + SHAPE). When using sized sprites, the value of SHAPE must be a multiple of 4.
- 2 COLOR for frame 1
- 3 SHAPE for frame 1
- :
- 2n COLOR for frame n
- 2n+1 SHAPE for frame n

*** GENERATORS - cartridge ROM

All the ROMed pattern generators must be grouped together in a contiguous block (starting at location GENERATORS). Each pattern generator must be 8 bytes long, and the number of pattern generators must conform to the value stored in NUMGEN:

byte:

- 0 bb,bb,bb,bb,bb,bb,bb,bb - the first 8 byte pattern generator (bb = binary graphic data)
- 8 bb,bb,bb,bb,bb,bb,bb,bb - the second 8 byte pattern generator
- etc. for a total of 8*NUMGEN bytes

*** STATUS - CPU RAM

An object's status area consists of 4 elements:

Byte:

- 0 1 byte for FRAME - indicates which of the object's frames is to be displayed.
- 1 2 bytes for X_LOCATION
- 3 2 bytes for Y_LOCATION - X_LOCATION and Y_LOCATION give the coordinate position of the upper left corner of the object on a "metaplane" which includes the pattern plane (see fig. 1). The origin of the pattern plane is coincident with the origin of the metaplane. The values at X_LOCATION and Y_LOCATION are 16 bit signed numbers which permits the positioning of an object anywhere within or outside of the pattern plane. This enables an object to be bled on or off the pattern plane in any direction.
- 5 1 byte for NEXT_GEN - an index which points to the next generator location which can be used when adding new generators to an object's generator tables. After the object's ROMed generators have been moved to its VRAM tables, ACTIVATE will set NEXT_GEN to equal FIRST_GEN_NAME + NUMGEN.

Some objects may require generators which are essentially modified versions of other generators (e.g. a generator which represents the pattern of another generator which has been rotated). ROM space can be conserved by including only the "unmodified" generators in the graphics data and using system routines to generate the modified versions. NEXT_GEN points to the next free location in that object's generator table to which a modified generator may be added. When adding new generators in this manner, NEXT_GEN should be updated in order to prevent new generators overwriting old ones.

5.0 COMPLEX OBJECTS

Complex objects are aggregates of other "component" objects which may be of any type (including other Complex objects) except Mobile objects. This object type gives the game programmer the ability to combine several objects in order to create a higher order graphic entity. Complex objects may have multiple frames and may be moved about on the display in the same manner as any other object.

Before a Complex object can be displayed, all of its component objects must be activated. This can be done by activating the complex object itself (ACTIVATE will then process all the component objects). However, if any of the component objects are type Semi-Mobile and the VDP is operating in graphics mode I, then all the component objects must be individually activated. If ACTIVATE is not used, then in addition to moving the generators to VRAM, the FRAME variable in the STATUS area for each of the component objects must be initialized to 0.

Once all the component objects are activated, a Complex object may be placed on the display in the same manner as any other object type. The object's STATUS area is initialized with the desired frame number and position, and then PUT_OBJECT is called. PUT_OBJECT will then determine the correct frame number and position for all of the component objects and place them on the display.

Each frame of the Complex object points to a list of frame numbers and a list of offsets. The list of frame numbers specifies the frame number to be used for each of the component objects. The list of offsets specifies the positional offset of each of the component objects from the Complex object's X and Y_LOCATIONS.

5.1 DETAILED DESCRIPTION OF COMPLEX OBJECT DATA STRUCTURE

Complex objects are defined in cartridge ROM by:

- 1) COM_OB - the object's high level definition
- 2) GRAPHICS - which contains frame and offset parameters. For each frame of the Complex object, these parameters define the frame numbers for the component objects and the positional relationship between the component objects.

and in CPU ram by:

- 1) STATUS - which contains the frame and position variables for the object.

The following is a detailed description of the data structure:

*** COM_OB - cartridge ROM

The high level definition of a Complex object contains pointers to the GRAPHICS and STATUS areas for the object, and a list of addresses pointing to the high level definition of the component objects.

Byte:

0	address of GRAPHICS (the start of the ROMed graphics data for the object)
2	address of STATUS (the object's RAM status area)
4	address of the 1st component object high level definition
6	" " " 2nd " " " "
:	:
2n+2	" " " nth " " " "

*** GRAPHICS - cartridge ROM

The graphics segment of a Complex object can be thought of as divided into three sections. The first section contains the following:

Byte:

0	OBJ_TYPE - This byte is divided into two parts: LSN - specifies the object type and must be equal to 4 for complex objects. MSN - contains the number of component objects that make up the complex object.
1	pointer to FRAME_0 (list of frame numbers)
3	pointer to OFFSET_0 (list of offsets)
5	pointer to FRAME_1
7	pointer to OFFSET_1
:	:
2n+1	pointer to FRAME_n
2n+2	pointer to OFFSET_n

*** FRAME_0 ... FRAME_n - cartridge ROM or CPU RAM

Each frame of a complex object specifies the frame numbers to be used for each of its component objects. The frame numbers to be used for any given frame are arranged in a list; the first entry being the frame number for the first component, the second entry the frame number for the second component, etc. There may be as many frame lists as there are frames, or several or all frames may share the same frame list. For example, if the only difference between frames of a complex object were the positional relationship of the component objects, then one frame list would be sufficient. The format of the frame list is as follows:

Byte:

0	frame number for 1st component
1	" " " 2nd "
:	:
n-1	" " " nth "

*** OFFSET_0 ... OFFSET_n - cartridge ROM or CPU RAM

Each frame of a Complex object also must have an offset list. This list determines the position of each of the component objects with respect to the position of the Complex object (its X and Y_LOCATIONS). Each entry in the offset list has two components, a one byte X displacement followed by a one byte Y displacement. These displacements are unsigned 8 bit integers and are added to the 16 bit values contained in the Complex object's X and Y_LOCATIONS to form the coordinate position for each of the component objects. As with the frame lists, there may be as many offset lists as there are frames, or fewer if several or all frames use the same offset list. The format of the offset lists is as follows:

byte:

0	X	displacement for the 1st component object			
1	Y	"	"	1st	"
2	X	"	"	2nd	"
3	Y	"	"	2nd	"
:				:	
2n	X	"	"	nth	"
2n+1	Y	"	"	nth	"

*** STATUS - CPU RAM

The STATUS area for a Complex object is similar to the STATUS area for any other object type. The only difference is that the 5th byte, EXT_GEN, is not used.

byte:

0	FRAME - the value in this byte indicates the frame to be displayed.
1	X_LOCATION - a two byte value determining the X position of the Complex object on the display.
3	Y_LOCATION - a two byte value determining the Y position of the Complex object on the display.

6.0 ACTIVATE

Calling sequence:

```
LD HL,OBJECT_NAME  
SCF  
CALL ACTIVATE ; MOVES GENERATORS TO VRAM  
:
```

-or-

```
LD HL,OBJECT_NAME  
OR A  
CALL ACTIVATE ; DOESN'T MOVE GENERATORS TO VRAM  
:
```

Registers used:

Uses all registers

RAM Used (starting at WORK_BUFFER):

If the object type is Semi_Mobile, the VDP is operating in Graphics Mode II, and bit 4 of OBJ_TYPE = 1, then 8 bytes starting at WORK_BUFFER are used. Otherwise no RAM is used.

6.1 DESCRIPTION OF ACTIVATE ROUTINE

The primary purpose of the ACTIVATE routine is to move the pattern and color generators from an object's GRAPHICS data area to the locations in the VRAM pattern and color generator tables assigned to it. In addition ACTIVATE will initialize certain variables in the object's STATUS and OLD_SCREEN areas. How the routine actually functions is dependent on the type of object being "activated", the graphics mode being used, and the state of the carry flag.

1) Activating Semi-Mobile objects

If the third address pointer in the object's high level definition is less than 8000H, then that address is taken to be the address of an OLD_SCREEN area for that object. The first byte in the OLD_SCREEN area is initialized with an 80H. This value indicates that no data has yet been saved in OLD_SCREEN. When PUT_OBJECT sees this value it will not attempt to restore the contents of OLD_SCREEN to the display.

The first byte in the object's STATUS area, the FRAME variable, is set to 0.

The 6th byte in the object's STATUS area will be initialized with a value equal to FIRST_GEN_NAME + NUMGEN. Routines which add new generators to the object's pattern and color generator tables may access this byte to determine where to put them.

If the carry flag is set when ACTIVATE is called, then the object's pattern and color generators will be moved to the pattern and color generator tables in VRAM. If the carry flag is not set, then the generators will not be moved; in this case only the STATUS and OLD_SCREEN areas will be affected. This feature is used when several objects share the same generators. ACTIVATE needs to be called with the carry flag set for only one of the objects, to get the generators to VRAM, and the other objects are ACTIVATED with the carry flag not set to prevent the generators from being moved again. The generators are moved as follows:

a) In Graphics Mode I

All the pattern generators (the number of which is given by the NUMGEN entry) are moved to the pattern generator table in VRAM. The first generator is moved to a location within the table specified by FIRST_GEN_NAME (i.e. the VRAM address to which the first pattern generator is moved = pattern generator base address + 8 * FIRST_GEN_NAME). The others are loaded sequentially.

In Graphics Mode I the pattern generator table is divided into 32 eight-byte blocks. The color for each block of 8 generators is defined by one color generator byte. Therefore, ACTIVATE needs to determine the number of color generator bytes that must be moved to the VRAM color generator table. It does this in the following manner (see NOTE below):

The first pattern generator will require a color generator byte at FIRST_GEN_NAME/8 offset from the start of the color generator table. The last pattern generator will require a color generator byte at (FIRST_GEN_NAME + NUMGEN - 1)/8. Therefore, the total number of color generators needed will be (FIRST_GEN_NAME + NUMGEN - 1)/8 - FIRST_GEN_NAME/8 + 1. ACTIVATE will move this number of color generator bytes to the color generator table starting at VRAM address = color table base address + FIRST_GEN_NAME/8.

b) In Graphics Mode II

In this mode the pattern and color generator tables are divided into three sections. Each section contains the generators which will be displayed in the corresponding third of the pattern plane. A Semi-Mobile object needs to have its generators moved to one or more sections depending on which thirds of the pattern plane it may appear in. The MSN of OBJ_TYPE in the object's GRAPHICS data area indicates which sections of the generator tables the pattern and color generators should be moved to as follows:

if bit 7 = 1 then move generators to the 1st section
if bit 6 = 1 then move generators to the 2nd section
if bit 5 = 1 then move generators to the 3rd section
The starting location within each section to which the generators will be moved is specified by FIRST_GEN_NAME. The first pattern/color generator will be located at FIRST_GEN_NAME and the rest will be loaded sequentially (e.g. if the MSN of a Semi-Mobile object = 1010B, then the pattern generators will be moved to VRAM address = pattern base address + FIRST_GEN_NAME * 8, and also to VRAM address = pattern base address + 1000H + FIRST_GEN_NAME * 8).

Bit 4 of OBJ_TYPE indicates whether there are 8 or 1 color generator bytes per pattern generator. If this bit is 0, then ACTIVATE will expect 8 color generator bytes per pattern generator. If bit 4 is 1, then only one color generator byte will be expected. This byte will be used to fill all eight bytes of the appropriate color generator in VRAM. The location within the color table to which the generators are moved is determined in the same fashion as the pattern generators (see above).

2) Activating Mobile objects

The first byte of the object's OLD_SCREEN area is initialized with 80H.

The frame variable, the first byte in the object's STATUS area, is initialized to 0.

Bytes 6 and 7 of the object's STATUS area are initialized with the value NEW_GEN from the object's GRAPHICS data. This value is used as a pointer to the beginning of an area reserved for the addition of generators created at game-on time.

3) Activating Sprite objects

The frame variable, the first byte in the object's STATUS area, is initialized to 0.

Byte 6 of the object's STATUS area is initialized with the value FIRST_GEN_NAME + NUMGEN.

All the generators in the GRAPHICS data area are moved to the sprite generator table in VRAM. The first generator is moved to the location specified by FIRST_GEN_NAME and the rest are loaded in sequentially.

4) Activating Complex objects

The number of component objects to activate is determined by the value contained in the MSN of OBJ_TYPE, the first byte in the object's GRAPHICS data area. Following the pointer to the object's STATUS area

is a list of addresses. Each address in the list points to the high level definition for one of the component objects. ACTIVATE is called once for each of the component objects, with the state of the carry flag set to the condition it was in when the routine was initially called.

NOTE:

An error in the ACTIVATE routine shows up when ACTIVATEing Semi-Mobile objects, or Complex objects containing Semi-Mobile objects, and the VDP is operating in Graphics Mode 1. This error causes an incorrect number of color generator bytes to be moved to the color table in VRAM when FIRST_GEN_NAME of the object is 80H or greater. In these instances, the cartridge program will have to fulfill the functions of ACTIVATE, and move the pattern and color generators to VRAM itself. Initialization of the STATUS and OLD_SCREEN areas can still be done by ACTIVATE; make sure the carry flag is NOT set when calling ACTIVATE on such objects.

7.0 PUT_OBJECT

Calling sequence (for all object types except Mobile objects):

```
LD IX,OBJECT_NAME  
CALL PUT_OBJECT
```

Registers used:

Uses all registers

RAM used (starting at WORK_BUFFER):

1. Semi_Mobile
If OLD_SCREEN is not used, then no RAM area is used by PUT_OBJECT. If OLD_SCREEN is used, then the maximum space, in bytes, used by PUT_OBJECT will be equal to the number of pattern blocks in the largest frame (i.e. the largest value of $X\_EXTENT * Y\_EXTENT$) plus 4.
2. Mobile
See discussion of PUT_MOBILE below.
3. Sprite
4 bytes.
4. Complex
The amount of RAM used is equal to the largest amount used by any of its component objects.

7.1 DESCRIPTION OF PUT_OBJECT

PUT_OBJECT places a frame of an object on the display. Which frame is displayed and where it is placed on the display are specified in that object's STATUS area. When PUT_OBJECT is called, it will determine what type of object is to be "put" on the display and then branch to one of four subroutines designed to handle that particular object type. These subroutines are: PUT_SEMI, PUT_MOBILE, PUT_SPRITE and PUT_COMPLEX. Below is a description of each routine.

7.2 DESCRIPTION OF PUT_SEMI

As the name implies, this routine handles the display of Semi-Mobile objects. A frame for a Semi-Mobile object is put on the display by writing the list of generator names that define the frame into the pattern name table in VRAM. In addition, the names in the name table that are overwritten may optionally be saved and later restored to the name table when the object is moved, or removed from the display.

The following algorithm is used to place Semi-Mobile objects on the display:

IF the object does NOT have an OLD_SCREEN (i.e. if bit 15 of the OLD_SCREEN address in the object's high level definition is set), THEN

DISPLAY the frame indicated by FRAME in STATUS at location specified by X_LOCATION and Y_LOCATION

ELSE

IF 1st BYTE of OLD_SCREEN is NOT 80H (there is valid data in OLD_SCREEN), THEN

DISPLAY OLD_SCREEN, first 2 bytes in OLD_SCREEN give X and Y coordinates of upper-left corner (in pattern plane positions) of OLD_SCREEN frame, third and fourth bytes give X and Y_EXTENTS of OLD_SCREEN frame

READ background pattern names over which frame of object will be displayed and save in OLD_SCREEN along with X and Y positions and X and Y_EXTENTS

DISPLAY new frame of object at called for X and Y position

ENDIF

7.3 PUT_MOBILE

Calling sequence:

```

:
LD IX,OBJECT_NAME
LD B,MODE                ; MODE = 0-3, DETERMINES THE METHOD
                           ; FOR COMBINING BACKGROUND AND OBJECT
CALL PUT_MOBILE          ; PUT_MOBILE ENTRY IS IN CARTRIDGE ROM
:                         ; (SEE DISCUSSION BELOW)
```

Registers used:

Uses all registers

RAM Used (starting at WORK_BUFFER):

In graphics mode I, 141 bytes
In graphics mode II, 203 bytes

7.4 DESCRIPTION OF PUT_MOBILE

This routine will place the specified frame of a Mobile object on the display. The location of the object and the frame to be displayed are determined by the variables FRAME, X_LOCATION and Y_LOCATION in its STATUS area. The X and Y_LOCATION variables specify the pixel position of the upper left corner of the object on the meta-plane. The object therefore may be displayed as entirely on the pattern plane or as bleeding on or off the pattern plane in any direction.

In general, PUT_MOBILE produces the image of a Mobile object on the display by creating a new set of pattern and color generators which depict the object superimposed on the background. Since all frames of Mobile objects will be 2 by 2 pattern blocks in size, the number of new generators which need to be created is 9. These 9 generators constitute a "surround" for the object within which the object will be displayed.

The 9 pattern generators which constitute the surround may be thought of as an array of 3 by 24 "surround" bytes. Similarly, the pattern generators which constitute the frame of the object to be displayed may be thought of as an array of 2 by 16 "frame" bytes. PUT_MOBILE uses one of two methods for combining these two arrays of generator bytes. Which method is used is selected by the state of bit 0 in register B when PUT_OBJECT is called.

If this bit is zero then an "additive" method is used to combine the object's graphics with that of the surround. In this method the "1" bits in the object's pattern generators are "moved" into the appropriate locations in a new set of surround generator bytes; these create a graphic of the Mobile object superimposed on the background. New surround color generator bytes are created as follows:

For each byte, if the corresponding pattern generator byte has not been changed (i.e. no "1" bits have been moved to it) then that color byte is left alone. If the corresponding pattern generator byte has had one or more "1" bits added to it, then the color portion of that byte is changed to the color of the Mobile object. Therefore, any parts of the background graphic represented by "1" bits will take on the color of the Mobile object when both the background and the object have "1" bits within the same pattern generator byte. The overall effect of this method is to maintain the "structural" integrity of the background pattern, even though the color of the background graphic will change to that of the object whenever elements of the object and the background exist within the same generator byte. Figure 2 illustrates the graphic effect of this mode of operation.

If bit 0 of register B is set to 1, then another method is applied. As in the above method, the "1" bits of the object's pattern generator bytes are again moved to the appropriate locations within the new set of surround generator bytes. However, surround generator bytes to

which bits are to be added will be cleared first (i.e. any elements of the background graphics which exist within the same byte as object graphics will be lost). The color generators are treated in the same manner as above. This gives the effect of the Mobile object "breaking holes" in the background as it moves through a pattern. See figure 3 for an illustration of this mode.

The limitation of only being able to display two colors within the same line of a pattern position force the above compromises when combining object and background generators. Which method should be used in a given situation will depend on the graphics and color of both the background and the object. Experimentation will be necessary to determine which method produces the least disruption of the background graphics.

Another option gives the programmer control over the choice of color0 for any color generator bytes which have had their color1 changed to the object's color.

If bit 1 of register B is 0, then the color0 of the above-mentioned color bytes will not be changed.

If bit 1 of register B is 1, then the color0 will be changed to transparent (thereby causing the backdrop color to be displayed as the color0).

Figures 4 and 5 illustrate these last two modes of operation respectively.

The new pattern and color generators are moved to the object's generator tables as follows:

Each Mobile-Object will have table space assigned to it (within the VRAM pattern and color generator tables) for 18 pattern and color generators. These tables are divided into an upper and a lower half. When a Mobile-Object is displayed, the "active" generators (i.e. those currently being used to display the object) will reside in either the upper or lower half of its generator tables. The half which is not in use is indicated by bit 7 of FRAME in that object's status area. This bit will be 0 when the lower half is not in use and 1 when the upper half is not in use. PUT_MOBILE moves the new generators to the half of the table not in use and also complements bit 7 of FRAME. This procedure enables the new generators to be moved to VRAM without disturbing the current display. If this were not done, it would be possible for one TV field to display partially updated generators and thereby cause the object to flicker.

Once the new set of pattern and color generators are moved to the VRAM pattern and color tables, PUT_MOBILE displays the Mobile-Object by writing the names of these new generators to the pattern name table in order to display the object at the called for location. In addition

the pattern names which are overwritten in the process of displaying the Mobile-Object are saved, and then restored to the name table, if necessary, when the object moves.

NOTE:

Due to an error near the beginning of the PUT_MOBILE routine, the beginning part of PUT_MOBILE will have to be included as part of the cartridge program. Instead of calling PUT_OBJECT for mobile objects, it will be necessary to call the version of PUT_MOBILE which will reside in cartridge ROM. This has the following two side effects:

1. Mobile-Objects may not be components of a Complex object.
2. The deferred write condition will not be recognized by

PUT_MOBILE.

The following is the section of PUT_MOBILE which must be incorporated as part of the cartridge program:

```
WORK_BUFFER      EQU      8006H
FLAGS            EQU      3
FRM              EQU      4
YDISP           EQU      0
XDISP           EQU      1
YP_BK           EQU      18
XP_BK           EQU      17
PX_TO_PTRN_POS  EQU      07E8H
GET_BKGRND      EQU      0898H
PM2             EQU      0AE0H
```

```
PUT_MOBILE      LD  IY,[WORK_BUFFER]      ; IY := WORK_BUFFER
```

```
    if in GRAPHICS MODE I
```

```
        RES 7,B
```

```
    if in GRAPHICS MODE II
```

```
        SET 7,B
```

```
        LD [IY+FLAGS],B
```

```
        PUSH HL
```

```
        LD H,[IX+3]
```

```
        LD L,[IX+2]
```

```
        LD A,[HL]
```

```
        LD [IY+FRM],A
```

```
        XOR 80H
```

```
        LD [HL],A
```

```
        INC HL
```

```
        LD E,[HL]
```

```
        LD A,E
```

```
        AND 7
```

```
        NEG
```

```
        ADD A,8
```

```
        LD [IY+XDISP],A
```

```
        INC HL
```

```
        LD D,[HL]
```



```

CALL PX_TO_PTRN_POS
LD [IY+XP_BK],E
INC HL
LD E,[HL]
LD A,E
AND 7
LD [IY+YDISP],A
INC HL
LD D,[HL]
CALL PX_TO_PTRN_POS
LD [IY+YP_BK],E
LD HL,[WORK_BUFFER]
LD DE,YP_BK+1
ADD HL,DE
LD D,[IY+YP_BK]
LD E,[IY+XP_BK]
LD BC,303H
PUSH IX
CALL GET_BKGRND
POP IX
JP PM2

```

The calling sequence for Mobile-Objects is:

```

LD IX,HIGH_LEVEL_DEFINITION
LD HL,GRAPHICS
LD B,MODE                      ; see above discussion for
                                ; parameter passed in reg B
CALL PUT_MOBILE

```

An additional patch is needed when operating in GRAPHICS MODE I. In this case the last instruction in the above code (JP PM2) is replaced with the following code:

```

THREE_GEN
PUSH IX                      ; Save another copy of object pointer
CALL PM2                     ; Call rest of OS PUT_MOBILE routine
POP IX                       ; Restore object pointer
LD IY,3                      ; Set up for 3 item VRAM write
LD A,[IX+6]                  ; Get FIRST_GEN_NAME
LD B,A                       ; And save another copy
AND A,7                      ; Evaluate MOD 8
CP 7                         ; If NE 7 then
JR NZ,THREE_GEN              ; 3 generators to move
LD IY,4                      ; Else, move 4 generators
LD A,B                       ; A := FIRST_GEN_NAME
SRL A                        ; Divide by 8
SRL A                        ; to get index into
SRL A                        ; color table
LD E,A                       ; DE gets pointer to object's
LD D,0                       ; color gens in VRAM
LD HL,W_BUF+88H              ; Point to 4th gen
PUSH HL                      ; Save pointer

```

```
LD A,[HL]
LD B,3          ; Copy this generator 3 times
COPY3 INC HL
LD [HL],A
DJNZ COPY3
POP HL          ; Get back pointer
LD A,4          ; Code for color table
CALL PUT_VRAM
RET
```

W_BUF is the address of the work area for OS routines (i.e. the address stored at WORK_BUFFER, 8006H, in cartridge ROM).

NOTE: In applications where the background color (color0) of the patterns over which the mobile object is to move and the color of the mobile object itself does not change, the above patch will not be needed. Instead it is sufficient to initialize the color generators for the mobile object to the desired color1/color0 combination.

7.5 PUT_SPRITE

This routine handles the display of all sprite objects; size0, size1, magnified and unmagnified.

The routine uses SPRITE_INDEX in the object's high level definition to determine which of the 32 sprites to use to implement the object. Positional information from X and Y_LOCATION in the object's STATUS area is used to update the vertical and horizontal position entries in the sprite attribute table in VRAM. The routine facilitates Sprite objects bleeding off to the left as well as completely off the screen by making use of the early clock bit.

Frame information for Sprite objects comes from the FRAME_TABLE in the object's GRAPHICS data area. Each frame is specified by a COLOR and a SHAPE byte. The SHAPE byte is added to the object's FIRST_GEN_NAME (the second entry in the object's GRAPHICS data area) and this sum is then moved to the name entry position in the sprite attribute table. Bit 7 of the COLOR byte is modified to reflect the required state of the early clock bit and then moved to color entry in the sprite attribute table.

7.6 PUT_COMPLEX

A Complex object is a collection of "component" objects. Each frame of a Complex object specifies the positional relationship and frame to be used for each of the component objects. The positional relationship for all the component objects is defined in an offset list and the frame for each component is defined in a frame list. There is one offset list and one frame list for each frame of the Complex object.

PUT_COMPLEX takes the following steps to display Complex objects:

1. Using the FRAME_LIST for the particular frame of the Complex object to be displayed, the frame of each of the component objects is updated.
2. X and Y_LOCATION for each of the component objects is formed by adding the X and Y offsets for each component, as specified in the OFFSET_LIST, to the X and Y_LOCATIONS from the Complex object's STATUS area. Note: the offsets are 8 bit unsigned numbers, so the location of the component objects will always be to the right and/or below the coordinate position indicated by X and Y_LOCATION in the Complex object's STATUS area.
3. Each of the component objects is displayed by calling PUT_OBJECT for each of the component objects.

8.0 DATA STRUCTURE SUMMARY FOR SEMI_MOBILE OBJECTS

*** ROM DATA AREAS ***

High level definition for a Semi-Mobile object; the address of the high level definition (SMD) is passed to the graphics routines when called on to process the object:

SMD	DEFW	GRAPHICS	;pointer to graphic data for object
	DEFW	STATUS	;pointer to status area
	DEFW	OLD_SCREEN	;pointer to save area for OLD_SCREEN

Graphics for semi_mobile objects are defined as follows:

GRAPHICS	DEFB	OBJ_TYPE	;LSN = 0, MSN used only in Graphics Mode II ;MSN bits 5,6,7 indicate which thirds of ;generator tables to move generators to ;if bit 7 then move gens to upper third ;" " 6 " " " " middle " ;" " 5 " " " " lower " ;bit 4 indicates how many color generator ;bytes per pattern generator to expect, if ;bit 4 = 0 then 8 bytes/gen else 1 byte/gen
	DEFB	FIRST_GEN_NAME	;index into VRAM tables where object's ;generators are located
	DEFB	NUMGEN	;number of ROMed generators for object
	DEFW	GENERATORS	;pointer to first of ROMed patterns
	DEFW	FRAME_0	;pointer to first frame data
	DEFW	FRAME_1	;pointer to second frame
	:	:	
	:	:	
	DEFW	FRAME_n	;pointer to Nth frame

The data for each frame is organized as follows:

FRAME_n	DEFB	X_EXTENT	;X_EXTENT and Y_EXTENT are the width and
	DEFB	Y_EXTENT	;height of the frame in pattern plane positions
	DEFB	NAME_0	;NAME_0 through NAME_n are the names of the
	DEFB	NAME_1	;patterns used for this frame. There must
	:	:	;exactly X_EXTENT * Y_EXTENT names and they
	DEFB	NAME_n	;must be arranged in a ROW major array, as
			;they are to be displayed on the screen (i.e.
			;the pattern representing NAME_0 will appear
			;at the upper-left corner of the frame, NAME_n
			;will appear at the lower-right corner).

The pattern generators are stored as follows:

GENERATORS	DEFB	bb,bb,bb,bb,bb,bb,bb,bb	;where bb = binary graphic data
	DEFB	bb,bb,bb,bb,bb,bb,bb,bb	
	:	:	
	:	:	
	DEFB	bb,bb,bb,bb,bb,bb,bb,bb	

Each group of 8 bytes corresponds to one generator. These generators are all moved to VRAM by ACTIVATE. The first generator will be located at FIRST_GEN_NAME (in Graphics Mode II, the MSN of OBJ_TYPE indicates which thirds of the generator tables will be initialised).

There are three possible formats for color generators. The first two are used in Graphics Mode II, the third in Graphics Mode I:

GRAPHICS MODE II:

1. If bit 4 of OBJ_TYPE is 0, then there must be 8 color generator bytes per pattern generator. The MSN of each byte specifies color1 for the corresponding pattern generator byte. The LSN specifies color0. (i.e. if the first color generator looks like: DEFB 1F,1F,1F,1F,39,39,39,39 , then the first 4 lines of the corresponding generator will have a color1=BLACK and color0=WHITE and the last 4 lines will have color1=LIGHT GREEN and color0=LIGHT RED.)

```
DEFB    cc,cc,cc,cc,cc,cc,cc,cc ;Color generator for 1st pattern
DEFB    cc,cc,cc,cc,cc,cc,cc,cc ; "      "      "      2nd  "
:
:
:
DEFB    cc,cc,cc,cc,cc,cc,cc,cc ; "      "      "      last  "
```

2. If bit 4 of OBJ_TYPE is 1, then only one color generator byte per pattern generator will be expected. This byte will be duplicated 8 times by the ACTIVATE routine when moving the generators to VRAM. Each byte has the same format as above. There must be one byte per pattern generator.

```
DEFB    cc      ;Color generator for 1st pattern
DEFB    cc      ; "      "      "      2nd  "
:
:
:
DEFB    cc      ; "      "      "      last  "
```

GRAPHICS MODE I:

3. In Graphics Mode I the pattern generator table can be thought of as divided up into 32 groups of 8 generators. Each group of pattern generators share the same color generator byte. Therefore, there must be one color generator byte for each group which contains pattern generators for the object.

*** RAM DATA AREAS ***

The status area for semi_mobile objects is as follows:

STATUS	DEFS	1	;Frame number to be displayed
	DEFS	2	;X_LOCATION, low byte first
	DEFS	2	;Y_LOCATION, low byte first
			;X and Y_LOCATION used by game program to
			;position object on screen and are 16 bit
			;signed numbers
	DEFS	1	;NEXT_GEN, index to area for adding new
			;generators

Old_screen data has the following structure (if OLD_SCREEN < 7000H, then it is in VRAM, else it is in CRAM):

OLD_SCREEN	DEFS	1	;X_PAT_POS, column in which the upper left corner ;of saved screen lies. This byte initialized to ;80H by ACTIVATE to indicate no data saved yet.
	DIFS	1	;Y_PAT_POS, row in which upper left corner of ;saved screen lies.
	DIFS	1	;X_EXTENT of saved screen
	DEFS	1	;Y_EXTENT of saved screen
	DIFS	n	;Where n = the largest screen that will need to ;be saved (i.e. the largest value of X_EXTENT * ;Y_EXTENT for any of the frames of this object).

9.0 DATA STRUCTURE SUMMARY FOR MOBILE OBJECTS

*** ROM DATA AREAS ***

High level definition for Mobile objects; the address of this data block (MOB) is passed to the various graphics routines when processing the object:

```
MOB      DEFW  GRAPHICS  ;pointer to graphic data for object
        DEFW  STATUS   ;pointer to status area
        DIFW  OLD_SCREEN ;pointer to save area for OLD_SCREEN
        DEFB  FIRST_GEN_NAME ;index into VRAM tables where object's
                                ;"active" generators are located, i.e.
                                ;those being used to display the object
                                ;space for 18 generators must be reserved
                                ;for each Mobile object
```

Graphics for a mobile object are defined as follows:

```
GRAPHICS  DEFB  OBJ_TYPE  ;LSN = 1
          DEFB  NUMGEN    ;number of ROMed generators for object
          DIFW  NEW_GEN    ;pointer to space for creation of new
                           ;generators
          DIFW  GENERATORS ;pointer to first of ROMed patterns
          DIFW  FRAME_0    ;pointer to first frame data
          DIFW  FRAME_1    ;pointer to second frame
          :
          :
          DIFW  FRAME_n    ;pointer to Nth frame
```

The data for each frame is organized as follows:

```
FRAME_n  DEFB  NAME_A,NAME_B,NAME_C,NAME_D,COLOR
                                ;where the pattern
                                ;for NAME_A will appear in the upper left
                                ;corner, NAME_B in the lower left, NAME_C
                                ;in the upper right and NAME_D in the lower
                                ;right. The MSN of COLOR specifies the
                                ;color for the frame (e.g. if MSN=7 then
                                ;frame will be CYAN) the LSN must = 0.
```

The pattern generators are stored as follows:

```
GENERATORS  DEFB  bb,bb,bb,bb,bb,bb,bb,bb ;where bb = binary graphic data
          DEFB  bb,bb,bb,bb,bb,bb,bb,bb
          :
          :
          DEFB  bb,bb,bb,bb,bb,bb,bb,bb
```

Each group of 8 bytes corresponds to one generator. Each generator is referenced by its position in the table. The value of the first generator's name is 0, the value of the second generator's name is 1 etc. New generators created at game-on time and stored at (NEW_GEN) continue in the numbering sequence. (i.e. if there are 8 generators in ROM, numbers 0-7, then generator number 8 refers to the first generator in a table starting at (NEW_GEN), etc.)

*** RAM DATA AREAS ***

The structure for a mobile object's status is as follows:

STATUS	DEFS	1	;the lower 7 bits specify the frame to be ;displayed, bit 7 indicates which VRAM table ;area is in use and must not be altered when ;changing the frame number
	DEFS	2	;X_LOCATION, low byte first
	DEFS	2	;Y_LOCATION, low byte first
			;X and Y_LOCATION used by game program to ;position object on screen and are 16 bit ;signed numbers
	DEFS	2	;NEW_GEN, points to area for adding new ;generators

Old_screen data has the following structure (if OLD_SCREEN < 7000H, then it is in VRAM, else it is in CRAM):

OLD_SCREEN	DEFS	1	;X_PAT_POS, column in which the upper left ;corner of saved screen lies. This byte ;is initialised to 80H by ACTIVATE to indicate ;that there is no data saved yet.
	DEFS	1	;Y_PAT_POS, row in which upper left corner of ;saved screen lies.
	DEFS	9	;Nine background names, arranged in ROW major ;order.

10.0 DATA STRUCTURE SUMMARY FOR SPRITE OBJECTS

*** ROM DATA AREAS ***

High level definition for Sprite objects; the address of this data block (SP_OBJ) is passed to the various graphics routines when processing the object:

```
SP_OBJ      DEFW  GRAPHICS      ;pointer to graphic data for object
            DEFW  STATUS        ;pointer to status area
            DEFB  SPRITE_INDEX  ;sprite number for this object (0-31)
```

Graphics for a sprite object are defined as follows:

```
GRAPHICS    DEFB  OBJ_TYPE      ;OBJ_TYPE = 3
            DEFB  FIRST_GEN_NAME ;name of first sprite generator
            DEFW  GENERATORS     ;pointer to ROMed generators
            DEFB  NUMGEN         ;number of ROMed generators for object
            DEFW  FRAME_TABLE    ;pointer to table of frame data
```

The data for each frame is organized as follows:

```
FRAME_TABLE DEFB  COLOR          ;sprite's color for this frame
            DEFB  SHAPE          ;offset from FIRST_GEN_NAME (generators
                                ;to be used for this frame)

            DEFB  COLOR          ;second frame color
            DEFB  SHAPE          ;second frame generator
            :
            :
            DEFB  COLOR          ;last frame color
            DEFB  SHAPE          ;last frame generator
```

The pattern generators are stored as follows:

```
GENERATORS  DEFB  bb,bb,bb,bb,bb,bb,bb,bb ;where bb = binary graphic data
            DEFB  bb,bb,bb,bb,bb,bb,bb,bb
            :
            DEFB  bb,bb,bb,bb,bb,bb,bb,bb
```

*** RAM DATA AREAS ***

The structure for a mobile object's status is as follows:

```
STATUS      DEFS  1              ;Frame number to be displayed
            DEFS  2              ;X_LOCATION, low byte first
            DEFS  2              ;Y_LOCATION, low byte first
                                ;X and Y_LOCATION used by game program to
                                ;position object on screen and are 16 bit
                                ;signed numbers
            DEFS  1              ;NEXT_GEN, index of free space in
                                ;generator table
```

11.0 DATA STRUCTURE SUMMARY FOR COMPLEX OBJECTS

*** ROM DATA AREAS ***

High level definition for Complex objects; the address of this data block (COM_OB) is passed to the various graphics routines when processing the object:

COM_OB	DEFW	GRAPHICS	;pointer to graphic data for object
	DEFW	STATUS	;pointer to status area
	DIFW	OBJECT_1	;pointer to component object 1
	DEFW	OBJECT_2	; " " " 2
	:	:	
	DEFW	OBJECT_n	; " " " n

Graphics for a Complex object are defined as follows:

GRAPHICS	DEFB	OBJ_TYPE	;MSN of OBJ_TYPE = number of component objects (must equal number of high level object addr)
			;LSN of OBJ_TYPE = 4
	DIFW	FRAME_LIST_0	;pointer to frame list for first frame
frame	DEFW	OFFSET_LIST_0	;pointer to list of offsets for first
	:	:	
	:	:	
	DEFW	FRAME_LIST_n	;pointer to frame list for last frame
frame	DEFW	OFFSET_LIST_n	;pointer to list of offsets for last

Each frame of a complex object is defined by a list of frame numbers and another list of offsets. Each entry in the FRAME_LIST specifies the frame to be displayed for one of the component objects. The first entry specifies the frame number for the first component object, the second entry specifies the frame number for the second, etc.

FRAME_LIST_n	DEFB	f1	;frame number for first component
	DEFB	f2	;frame number for second component
	:	:	
	:	:	
	DEFB	fn	;frame number for last component

The OFFSET_LIST specifies the location of each of the component objects with respect to the X and Y_LOCATION given in the complex object's STATUS area as follows:

```

OFFSET_LIST_n  DEFB    Xdisp      ;Xdisp = amount first component displaced
object          .        ;horizontally from X_LOCATION of complex
                DEFB    Ydisp      ;Ydisp = amount first component displaced
object          .        ;vertically from Y_LOCATION of complex
                DEFB    Xdisp      ;same for second component
                DEFB    Ydisp
                :
                :
                DEFB    Xdisp      ;same for last component
                DEFB    Ydisp
  
```

*** RAM DATA AREA ***

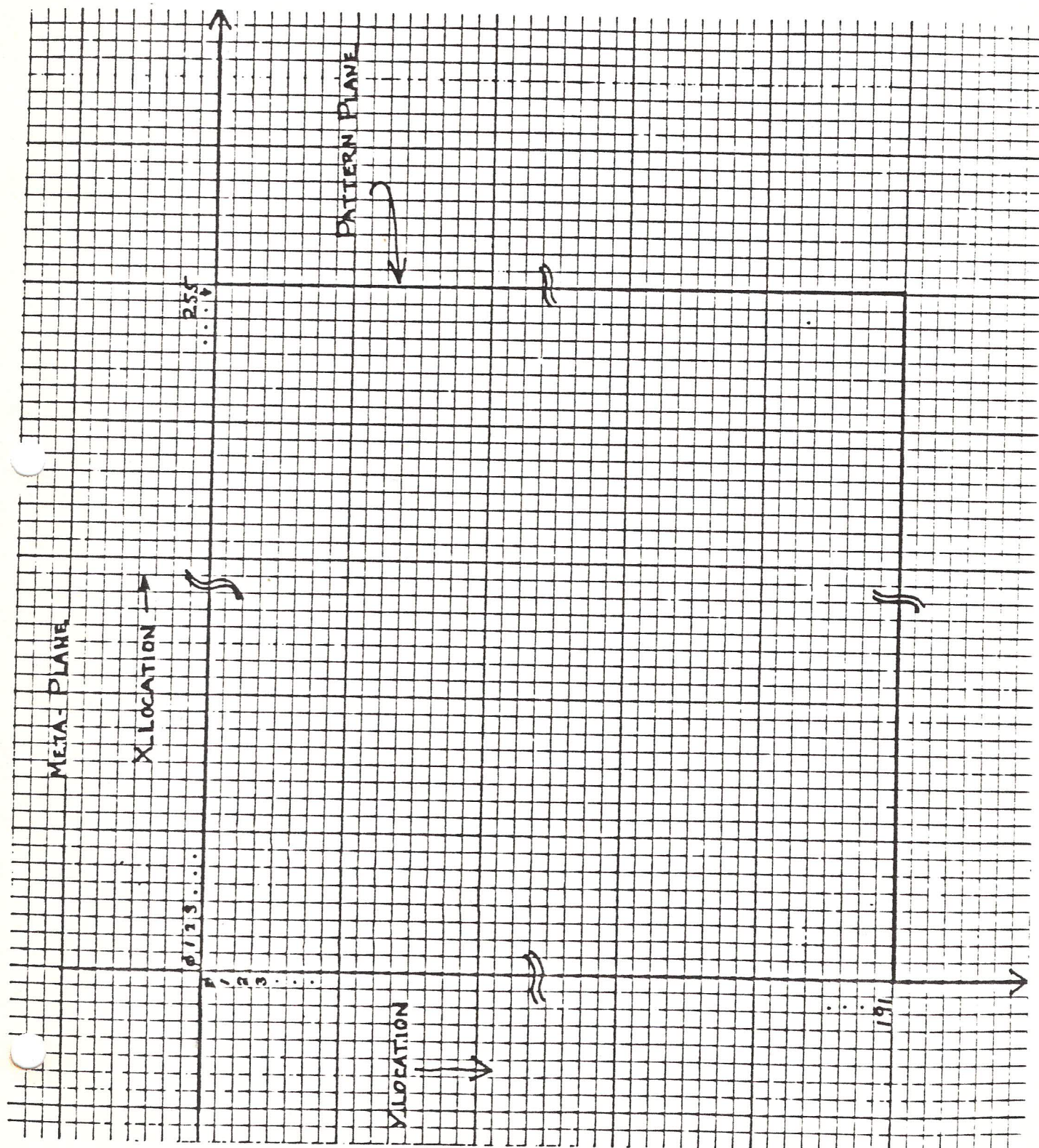
Status area for a Complex object:

```

STATUS        DEFS    1          ;Frame number to be displayed
               DEFS    2          ;X_LOCATION of object
               DEFS    2          ;Y_LOCATION of object
  
```


Figure 1

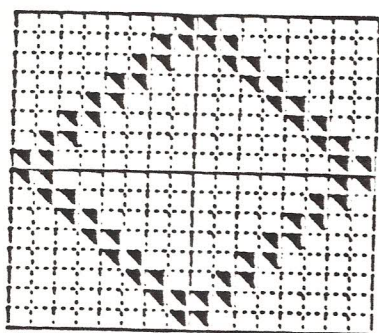
This figure illustrates the relationship between the Meta-Plane and the Pattern Plane. X_LOCATION and Y_LOCATION are sixteen bit signed variables which determine the location of an object's upper-left corner. These variables are part of each object's STATUS area in CPU RAM.



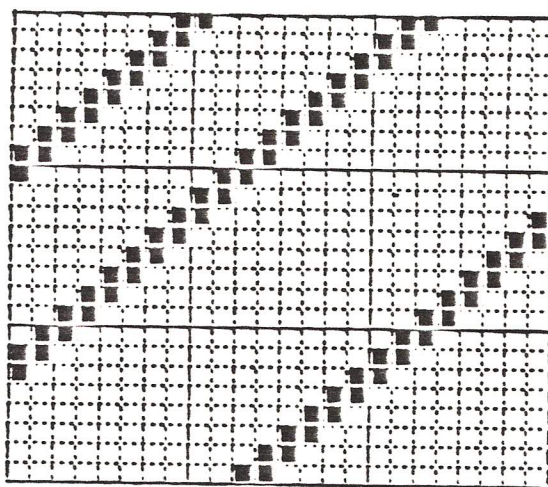
The following figures represent the four methods which PUT_MOBILE uses to superimpose a Mobile object upon a background. A parameter passed to PUT_MOBILE in register B, selects which of the four methods will be used.

Legend:

-  represents the color0 of the background
-  represents the color1 of the background
-  represents the color of the Mobile object
-  represents the backdrop color



Mobile object.



Three by three pattern position "surround"
 onto which the Mobile object will be placed.

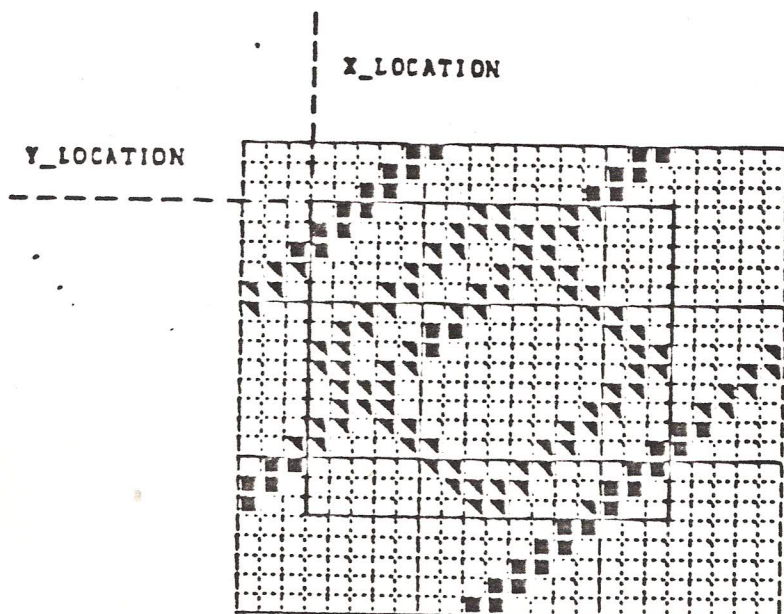


Figure 2: Mobile object
 superimposed on surround
 when parameter passed in
 register B = 0.

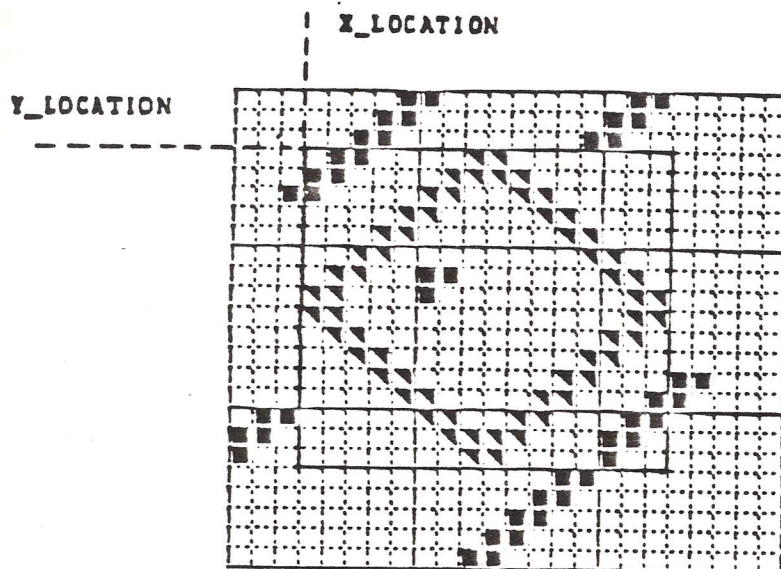


Figure 3: Mobile object superimposed on surround when parameter passed in register B = 1.

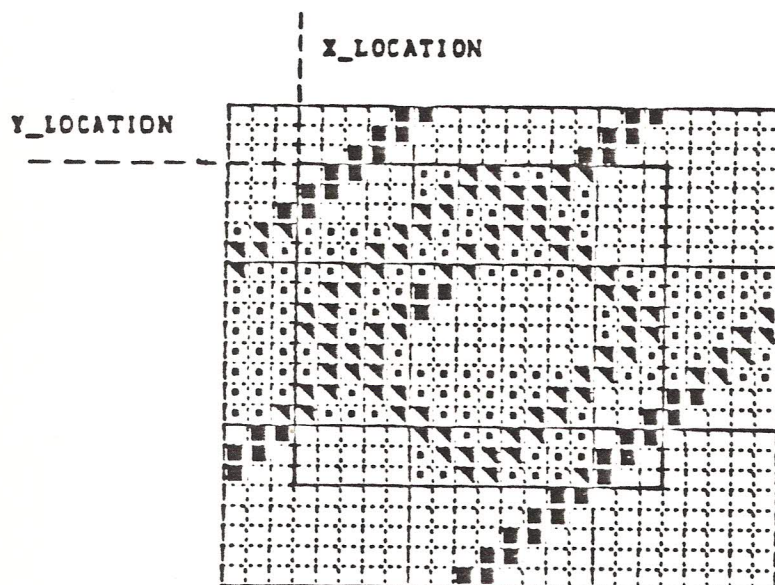


Figure 4: Mobile object superimposed on surround when parameter passed in register B = 2.

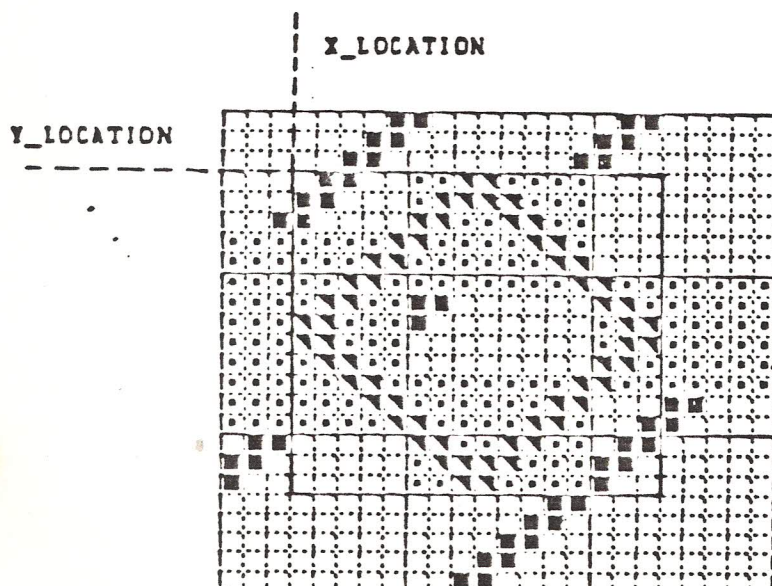


Figure 5: Mobile object superimposed on surround when parameter passed in register B = 3.

APPENDIX C

COLECOVISION
SOUND USERS' MANUAL

Version 1.1
May 3, 1982

CONFIDENTIAL
DO NOT COPY

*** CONTENTS ***

page	
1	SUMMARY OF FEATURES
2	GENERAL DESCRIPTION
2	Song data areas, SxDATA: RAM map mode of sound chip operation
2	Note list storage and note headers
3	LST_OF_SND_ADDRS and PTR_TO_LST_OF_SND_ADDRS
3	Hierarchy of song data areas: truncation, priority, overwriting
4	The four basic routines, briefly
7	NOTES
7	Terminology
8	Frequency sweeps and note duration
9	Attenuation sweeps
10	Rests
10	Type 0: fixed frequency, fixed attenuation
10	Type 1: swept frequency, fixed attenuation
10	Type 2: fixed frequency, swept attenuation
11	Type 3: swept frequency, swept attenuation
11	Noise notes: special case Type 2 notes
13	OPERATING SYSTEM ROUTINES
13	INIT_SOUND
13	ALL_OFF
14	JUKE_BOX
15	SND_MANAGER
16	PLAY_SONGS
17	FREQ_SWEEP
18	ATN_SWEEP
19	PROCESS_DATA_AREA
20	LOAD_NEXT_NOTE
21	UTILITIES
22	SPECIAL SOUND EFFECTS
22	Sound effects as notes within a song ("note effects")
24	A sound effect as a single sound
25	Pseudo code listing of main routines

figure

1	SxDATA
2	Note Header
3	LST_OF_SND_ADDRS
4	Rests and Type 0: fixed frequency, fixed attenuation
5	Type 1: swept frequency, fixed attenuation
6	Type 2: fixed frequency, swept attenuation
7	Type 3: swept frequency, swept attenuation
8	Noise notes: special case Type 2 notes
9	Dedicated cartridge RAM locations

The Colecovision operating system includes routines which allow the cartridge programmer to store "songs" and simple sound effects in tabular form in cartridge ROM, and play them on request during the game. More complex, special sound effects can also be created and played within the same data structure and procedural format. This Sound Users' Manual describes the data structures expected by and the use of the operating system sound routines.

SUMMARY OF FEATURES

- * six song note types: combinations of fixed or variable frequency and attenuation, "noise" (percussion) notes, and rests
- * hierarchical structure of local data areas assigned to each sound channel, which allows the temporary "overwriting" of lower priority songs (i.e., lower priority songs are not truncated by higher priority songs or sound effects that use the same channel, but continue unheard until the higher priority songs finish)
- * the ability to easily include a special sound effect (say, a cymbal crash) as part of a song composed primarily of musical tones
- * both songs and special, independent sound effects (e.g., an explosion sound) can utilize the same data structures and output procedures
- * sweep routines, which automatically create frequency or attenuation sweeps, simplify note data storage and can be used by special sound effects
- * song end codes allow songs to be played once, or automatically restarted upon completion (repeat forever)

GENERAL DESCRIPTION

* Song data areas, SxDATA: RAM map mode of sound chip operation

The Colecovision operating system provides sound routines which output frequency, attenuation, and control data to the TI 76489 sound chip. Data to be sent to a particular sound generator channel is expected to be stored within a ten byte block of CPU RAM called a "song data area". A song data area, then, contains a RAM record of the current values "playing" on a sound channel.

Each song data area can also contain timing and descriptive information which allows for simple generation of musical notes. A "song" can be created by storing in CART ROM a list of note parameters which specify note duration, frequency, and attenuation. When a song is started, O/S routines are provided to load the data describing the first note of the song into a song data area. Each song data area is then processed at regular intervals by routines which modify and output the area's data to the sound chip. When a note is completed, the next note in the song is automatically loaded and the process continues.

O/S routines also exist which facilitate the creation of "special effects": sound routines written by the cartridge programmer which algorithmically generate data to be sent to the sound chip (as opposed to the table look-up, song approach).

RAM space for at least four song data areas must be reserved by the cartridge programmer: one each to describe the current status of the four sound chip channels. More than four song data areas will be required if the ability to "overwrite" lower priority songs is desired, and some songs may share the same data area (see the following discussion, "Hierarchy of song data areas: priority, truncation, overwriting"). The first byte in each song data area, byte 0 (its offset from the beginning of the data block = 0), contains the channel number upon which the song is to be played (0: noise generator, 1 to 3: tone generator) and the song's identification number (SONGNO: 1 to 61). A song data area is referred to throughout the rest of this manual as "SxDATA", where x = the song's SONGNO. For a detailed description of each byte in a song data area, see the following discussion, "NOTES", and refer to Figure 1.

* Note list storage and note headers

A note list is a sequential list of frequency and timer data stored in cartridge ROM that, when processed and output to the sound chip, create the notes that comprise a song. Each block of 1 to 8 bytes of data that describes a note in the list must begin with a one byte header which contains information (bit flags or values) that indicate (see Figure 2):

- 1) The number of the channel upon which to play the note
- 2) The note type (one of 4 combinations of fixed or swept frequency and attenuation, plus a rest).

The single byte header can also be used as an end-of-song marker, a repeat-song indicator, or an indicator that a "note" is to be determined algorithmically by a special sound effect routine (the starting address of which immediately follows).

A 16 bit pointer to the location of the header of the next note to be played (NEXT_NOTE_PTR) is maintained by the O/S routine SND_MANAGER in each song's data area (SxDATA, offsets 1 and 2).

* LST_OF_SND_ADDRS and PTR_TO_LST_OF_SND_ADDRS

The O/S routines expect the ten byte long song data areas to be stored contiguously in CPU RAM, starting with the data area used by song number one. The beginning addresses of each of these data areas, as well as the addresses of the headers of the first note in each song, are stored in a ROM table called LST_OF_SND_ADDRS (see Figure 3): The cartridge programmer may place this table wherever desired in ROM. The O/S routines know its starting address through a dedicated CPU RAM location, PTR_TO_LST_OF_SND_ADDRS, which must be loaded with the 16 bit address of the table by the cartridge program before calling any O/S routines which use it (see description of INIT_SOUND).

* Hierarchy of song data areas: truncation, priority, overwriting

The routine that does the processing of the note data stored in the song data areas, SND_MANAGER, and the routine that outputs the modified data to the sound chip, PLAY_SONGS, are designed to be called by the cartridge program every Video Display Processor (VDP) interrupt (every 16.7 ms). Starting with the data area for song number one, SND_MANAGER processes the appropriate timer and sweep counters and modifies the frequency and attenuation data accordingly. If the data area is assigned to a special effect, SND_MANAGER calls that effect. When a note is finished, SND_MANAGER, using the data area's next note pointer, moves data for the next note of the song into the area.

After the operations upon a data area have been performed, the sound chip channel number (CH#) stored in byte 0 of that data area is consulted and the appropriate "channel data area pointer" (PTR_TO_S_ON_x) is set to point to the beginning of the data area just processed (four of these pointers exist at dedicated 16 bit locations in CPU RAM, one for each of the sound generator channels). The following data areas are then processed in the same fashion, in order of occurrence, until the end of data area code, 00, is reached. If a data area is inactive, i.e., if the song(s) that use it aren't playing at the moment, SND_MANAGER simply passes it over, doing no processing or channel data area pointer modification.

PLAY_SONGS, usually called immediately prior to SND_MANAGER, outputs data to the sound chip from the four song data areas pointed to by the channel data area pointers. Thus, a channel output priority is established on the basis of ordinal position within the data area block: the last data area processed that uses a given channel is the one that will be played on that channel. E.g.,

order of data area
within data block
containing all
song data areas

songs that use this data area:
SONGNO/CH# song is to be played on

1st	1/CH#x	(remember: although other songs may use this data area also, song 1 MUST use it)
...		
5th	6/CH#2	
6th	3/CH#2	
7th	11/CH#2	
...		
10th	2/CH#3; 4/CH#3; 7/CH#3	

First, consider channel 2. Let's say that the only songs which use channel 2 are assigned three contiguous song data areas, 5th through 7th (grouping songs which use the same channel isn't necessary as far as the code is concerned, but it makes it simpler to think about). SND_MANAGER, as it makes its way through the song data areas in order, will process the 5th data area (which "belongs" to song 6) and set PTR_TO_S_ON_2 to the address of byte 0 in the 5th data area. Then, the 6th area will be processed, resulting in resetting PTR_TO_S_ON_2 to the 6th data area (song number 3). Likewise, the 7th data area will be processed, finally leaving PTR_TO_S_ON_2 pointing to the 7th data area (song number 11). The next time PLAY_SONGS is called, it will send to sound chip channel 2 frequency and attenuation data in the data area pointed to by PTR_TO_S_ON_2, namely, the data for song number 11.

Note that although only song 11 will be heard on channel 2 this pass through PLAY_SONGS, the timers and data for songs 6 and 3 were nonetheless modified, regardless of the existence of the higher priority song 11. That is, all songs "keep going", whether or not their data will be output during PLAY_SONGS. Songs 6 and 3 are said to have been "overwritten" by song 11. Should song 11 become inactive (end) before song 6 and/or song 3, then the highest priority of the remaining active songs (i.e., the last data area within the block of data areas to use a given channel) will be heard.

Thus, assigning several data areas to songs which use the same channel allows the creation of "background" songs which can be momentarily interrupted (overwritten) by a higher priority song or sound effect (e.g., an explosion, or a bonus song) and continue on after the overwriting song is over.

Now examine the 10th song data area. Again, let's say that the only songs that use channel 3 are shown here and that they all share the 10th data area. In this case, the programmer may have arranged things such that songs 2, 4, and 7 are never active simultaneously: i.e., there was no reason to assign three different data areas for songs which never overlap. If, however, this is not the case and, say, song 4 may be started before song 7 is finished, the O/S routines would stop song 7 in favor of song 4. That is, for songs that share the same data area, the most recent song started is the song heard, and interrupted songs do not continue: i.e., songs sharing the same data area truncate each other. In many cases this may be both acceptable and desirable, as it saves RAM space.

NOTE: The preceding description states that a channel data area pointer is updated every time SND_MANAGER processes a song data area: this is actually not the case. To save processing time, a routine which updates all the pointers is called only when, after loading the next note in a song, SND_MANAGER detects that the data area's CH# or SONGNO has changed. This happens whenever the next note: 1) uses a different channel (see "Noise notes: special case Type 2 notes"), 2) is a special effect note, or 3) is an end-of-song indicator. It is only necessary to update the channel data area pointers in these cases, and when a new song is started (in JUKE_BOX). See "Pseudo code versions of main routines".

* The four basic routines, briefly

The following four O/S routines are the only ones that need be called to create songs which use the six standard note types (more complete descriptions of each routine can be found in the "OPERATING SYSTEM ROUTINES" section):

INIT_SOUND: This routine should be called immediately after power on, before any sound processing can occur. It turns off the sound generators, initializes the CART RAM locations to be used as song data areas, sets up the four channel data area pointers, and initializes PTR_TO_LST_OF_SND_ADDRS.

INPUT: n
TYPE: 8 bit constant
PASSED: in B
DESCRIPTION: number of song data areas used by the game

INPUT: LST_OF_SND_ADDRS
TYPE: 16 bit address
PASSED: in HL
DESCRIPTION: LST_OF_SND_ADDRS is the base address of a list of the starting addresses of each song's data area and note list.

OUTPUT:

- 1) turns off all sound generators .
- 2) initializes PTR_TO_LST_OF_SND_ADDRS
- 3) writes the inactive code (OFFH) to byte 0 of the n song data areas
- 4) stores 00 at end of song data areas
- 5) sets the 4 channel song pointers to a dummy inactive area
- 6) sets SAVE_CTRL to OFFH (see "Noise notes" discussion)

JUKE_BOX: JUKE_BOX is called to start a song. Using a song number passed in B, JUKE_BOX loads the data for the song's first note into the appropriate song data area, thereby truncating whatever song had been "playing" in that data area. (The address of the appropriate area is found by using the song number as an index into the LST_OF_SND_ADDRS table). It also formats the data area's header and sets up the next note pointer. If the song is a special sound effect, its next note pointer is set to the address of the special effect routine. The next time PLAY_SONGS is called, that song's first note will be played.

If JUKE_BOX is called with a song number of a song already in progress, it returns immediately (i.e., it doesn't restart the song).

INPUT: song number to be started
TYPE: 8 bit constant, 1 to 61
PASSED: in B

CALLS: PT_IX_TO_SxDATA, LOAD_NEXT_NOTE_PTR, UP_CH_DATA_PTRS

OUTPUT:

- 1) moves the song's first note data to the appropriate song data area
- 1) formats byte 0 header of the song's data area
- 2) points next note pointer in data area (bytes 1&2) to address of first note in song, or address of special sound effect routine

SND_MANAGER: SND_MANAGER should be called every VDP interrupt (every 16.7 ms). For each data area, SND_MANAGER processes the appropriate timer and sweep counters and modifies the frequency and attenuation data accordingly. If the data area is assigned to a special effect, SND_MANAGER calls that effect. When a note is finished, SND_MANAGER, using the data area's next note pointer, moves data for the next note of the song into the area. If SND_MANAGER reads a header byte (in CART ROM) that has bits 3&4 set, indicating repeat song, it will start the song again by reloading the first note in the song.

After the operations upon a data area have been performed, if necessary, the channel data area pointers (PTR_TO_S_ON_x) are updated. The following data areas are processed in the same fashion, in order of occurrence, until the end of data area code, 00, is reached.

SND_MANAGER does not output the modified frequency and attenuation data.
PLAY_SONGS is called just before SND_MANAGER to do this.

Special codes in byte 0 of the song data area indicate:

- 255: data area inactive, do no processing
- 62: a special effect is to be played; SND_MANAGER calls the effect routine
- 0: end of song data areas (SND_MANAGER processes data areas until it sees 0 in byte 0)

NOTE: Song number 1 MUST use the first area in the block of song data areas.

INPUT: none

CALLS: PT_IX_TO_SxDATA, PROCESS_DATA_AREA

OUTPUT: Calls routines which:

- 1) decrement song duration and sweep timers
- 2) modify swept frequency and attenuation values
- 3) call special effects routines where necessary
- 4) update the channel data area pointers if necessary
- 5) restart the song if indicated

PLAY_SONGS: PLAY_SONGS takes the frequency and attenuation data pointed to by the four channel data area pointers (PTR_TO_S_ON_x) and outputs it to the four sound chip generators.

INPUT: none

CALLS: TONE_OUT, UPATNCTRL

OUTPUT:

- 1) current freq and atn data is output to each tone generator, if song/effect on that channel is active; if song on that channel is inactive, that generator is turned off
- 2) noise generator is sent current atn data, and control data, if new
- 3) modifies SAVE_CTRL if necessary

These four routines would normally be called as follows:

power on inits done by O/S

cartridge program receives control:

LD B, # of song data areas used in the game

LD HL, address where LST_OF_SND_ADDRS is stored in ROM

CALL INIT_SOUND to initialize song data areas

whatever other power on inits you want to do

start game:

.

.

.

LD B, # of song you want to start

CALL JUKE_BOX to set up for start of song

.

.

VDP interrupt occurs:

CALL PLAY_SONGS to output data

CALL SND_MANAGER to process song data

whatever else you want to do during VDP interrupts

RETN to game

NOTES

* Terminology

Each note in a song has an associated 10 bit frequency (except for noise notes) and 4 bit attenuation which is output to the sound chip every time `PLAY_SONGS` is called. The initial frequency and attenuation values (stored in 2 bytes) are part of a block of 4 to 8 bytes that describe a single note within a song's ROM note list. The remaining bytes are used to indicate sound channel, note type, duration, and various timers and values associated with swept notes.

The following are explanations of names and symbols used throughout this manual to refer to bytes, or segments of bytes, within both a note's ROM note list and a RAM song data area (see Figure 1):

`Bx`: `x` is a symbol used to graphically separate bits or nibbles within a byte

`Bx`: means bit `x` of a byte, bit 7 being the most significant bit

byte `x`: refers to the offset of a byte within a data block, byte 0 being the first byte in the block

MSN, LSN: MSN means the most significant nibble of a byte, LSN is the least significant nibble

CH#: the sound channel upon which a note is to be played; 0 = noise generator, 1 to 3 = the 3 tone generators

SONGNO: the song number of the song playing in a song data area; 1 to 61; SONGNO 62 means a special sound effect is using the data area

NEXT_NOTE_PTR: 2 byte address of the ROM location of the data block for the next note to be played in a song

F0 - F9: the 10 bit frequency data to be sent to a sound chip tone generator; F0 is the most significant bit; see data sheets, TI 76489

ATN: 4 bit attenuation data to be sent to any of the four sound generators

CTRL: 3 bit control data for sound chip noise generator; FB NF0 NF1 (called SHIFT in this manual); see data sheets for details

NLEN: 1 byte that directly or indirectly determines the duration of a note

FPS: 4 bit frequency prescaler, used with NLEN to determine the length of a frequency sweep

FPSV: 4 bit temporary storage location for FPS; this variable gets decremented every VDP interrupt, and is reloaded from FPS

FSTEP: the size of a step in a frequency sweep, an 8 bit two's complement signed value that is added to the current 10 bit frequency at a rate determined by NLEN and FPS; 1 to 127, -1 to -128

ALEN: 4 bit number of steps in an attenuation sweep

ASTEPS: ASTEPS is the signed, 4 bit size of a step in an attenuation sweep; can take values 1 to 7, -1 to -8 (MSB = 1 means negative)

APS: 4 bit attenuation prescaler, used with ALLEN to determine the length of a frequency sweep

APSV: 4 bit temporary storage location for APS; this variable gets decremented every VDP interrupt, and is reloaded from APS

* Frequency sweeps and note duration

The time duration of a note = the number of passes by PLAY_SONGS through the note times 16.7ms (the VDP interrupt period). (note that the time the note is HEARD could be shorter: see "Attenuation sweeps" discussion) The number of PLAY_SONGS passes is always determined directly or indirectly by NLEN. NLEN, however, has two meanings, depending upon whether or not a note's frequency is swept:

Fixed frequency notes - In this case, NLEN is decremented every VDP interrupt and therefore directly determines the length of a note:

$$\text{duration} = \text{NLEN} * 16.7\text{ms}$$

NLEN should have values in the range 0 to 255 (0 => 256), giving a maximum duration of ~ 4.25 seconds.

Swept frequency notes - Here, the prescaler variable, FPSV, is decremented until zero before NLEN is decremented. Once FPSV goes to zero, it's reloaded from FPS; however, an initial value for FPSV, which enters into the calculation of the the length of the first step in the sweep, is stored in ROM along with the rest of the note's initial data, and it may be different from the reload value, FPS. For a frequency swept note:

$$\text{note duration} = [(\text{NLEN} - 1) * \text{FPS} + \text{initial FPSV}] * 16.7\text{ms}$$

So, NLEN again determines note duration, but in an indirect fashion (in concert with FPS and the initial FPSV).

In the case of a frequency swept note, NLEN can be thought of as one of four parameters that describe the sweep: 1) the starting frequency, 2) FSTEP, a signed step size, i.e., a delta frequency that is periodically added to the current frequency, 3) the prescaler value (FPS) which determines the length of time at any one frequency step, and 4) NLEN, the number of steps in the sweep.

The duration of each step in the sweep is given by the following:

$$\begin{aligned} \text{duration 1st step} &= \text{initial FPSV} * 16.7\text{ms} \\ \text{duration all others} &= \text{FPS} * 16.7\text{ms} \end{aligned}$$

Frequency sweep parameter ranges:

FSTEP - signed 8 bit two's complement number: 1 to 127, -1 to -128; an FSTEP of 0 tells FREQ_SWEEP that the note is not frequency swept, and the note's duration is determined by directly decrementing NLEN (prescaler is disregarded)

FPS - 4 bit frequency prescaler, used with NLEN to determine the length of a frequency sweep: 0 to 15 (0 => 16)

FPSV - 4 bit temporary storage location for FPS; this variable gets decremented every VDP interrupt, and is reloaded from FPS: 0 to 15 (0 => 16)

NLEN - 8 bit note duration for a fixed frequency note: 0 to 255 (0 => 256)
8 bit number of steps for a swept frequency note: 2 to 255 (0 => 256)

Note durations:

Fixed frequency -	NLEN * 16.7ms
Swept frequency -	[(NLEN - 1) * FPS] + initial FPSV * 16.7ms
duration 1st step =	initial FPSV * 16.7ms
duration all others =	FPS * 16.7ms

* Attenuation sweeps

Volume attacks and decays can be thought of as attenuation sweeps: a sweep from low to higher volume is an attack, a sweep in the other direction is a decay. Attenuation sweeps are created in a similar fashion to the frequency sweeps described above, the primary difference being that attenuation sweep parameters don't take on 8 bit values. The full volume range for the attenuation registers on the 76489 chip is 0 (ON) to 15 (OFF), so step sizes and number of sweep steps greater than 4 bits aren't generally useful.

Just as in the case of a frequency swept note, attenuation sweeps have four parameters that describe the sweep: 1) the starting attenuation, 2) ASTEP, a signed step size, i.e., a delta attenuation that is periodically added to the current attenuation, 3) the prescaler value (APS) which determines the length of time at any one attenuation step, and 4) ALEN, the number of steps in the sweep.

The prescaler parameters, APS and APSV, are the same size (4 bits) and mean exactly the same thing as their frequency counterparts. ALEN, the number of steps in the sweep, is only 4 bits (compared to an FLEN of 8 bits), but 15 steps of even the smallest step size (+/-1) can sweep a generator from full on to full off. ASTEP, in order to squeeze it into a nibble, has been limited to a 4 bit signed number (3 bits data, 1 bit sign). This gives a range of step values from 1 to 7, -1 to -8. This shouldn't be too limiting, since most attenuation sweeps are implemented with the smallest step size.

NOTE: Recall that, as far as SND_MANAGER is concerned, the length of a note is determined, directly or indirectly, only by NLEN, a timer/counter that is decremented during FREQ_SWEEP; i.e., the duration of a note is independent of what is happening to its attenuation. Therefore, the programmer should take care to see that an attenuation sweep isn't inadvertently created that ramps the volume down to off before SND_MANAGER, through NLEN, has decided that the note is over. However, since ATN_SWEEP simply leaves the attenuation alone once it's finished a sweep, the independence of attenuation sweep length and note length may be put to good use: e.g., a sforzando can be accomplished by making an attenuation sweep (to a still audible volume) end before the rest of the note.

ALEN, like NLEN for a frequency sweep, is the number of steps in an attenuation sweep and can take on values from 0 to 15. However, since a "step" consists of a tone (which may be frequency swept) played at a fixed attenuation level, a sweep of 1 step doesn't make sense. ALEN values, then, should range from 2 to 15. An ALEN value of 0 causes a sweep of sixteen steps (NOTE: ATN_SWEEP "wraps around" at 0 and 15, i.e., subtracting 1 from 0 results in 15, and adding 1 to 15 results in 0).

The duration of an attenuation sweep can be calculated as follows:

duration entire sweep = $[(\text{ALEN} - 1) * \text{APS}] + \text{initial APSV}] * 16.7\text{ms}$
 duration 1st step = initial APSV * 16.7ms
 duration all others = APS * 16.7ms

Attenuation sweep parameter ranges:

ALEN: 4 bit number of steps in an attenuation sweep; can take values from 2 to 15 (0 => 16 steps).

ASTEP: ASTEP is the 4 bit signed (two's complement) size of a step in an attenuation sweep; can take values from 1 to 7, -1 to -8.

APS: 4 bit attenuation prescaler, used with ALEN to determine the length of a frequency sweep: 0 to 15 (0 => 16).

APSV: 4 bit temporary storage location for APS; this variable gets decremented every VDP interrupt, and is reloaded from APS: 0 to 15 (0 => 16)

Descriptions of each of the six note types follow:

* Rests

See Figure 4.

byte 0: B5 set indicates a rest, to be played on CH# in B7 - B6
 duration = $(B4 - B0) * 16.7\text{ms}$, can take values 1 to 31

* Type 0: Fixed frequency, fixed attenuation

See Figure 4.

byte 0: header, CH# in B7 - B6, note type = 0 in B1 - B0
 byte 1: least significant 8 bits of the 10 bit frequency data (constant)
 byte 2: MSN = 4 bit ATN data (constant throughout the note)
 LSN = 0 0 F0 F1, the top 2 bits of the frequency data (constant)
 byte 3: NLEN, duration of the note = $NLEN * 16.7\text{ms}$

* Type 1: Swept frequency, fixed attenuation

See Figure 5.

byte 0: header, CH# in B7 - B6, note type = 1 in B1 - B0
 byte 1: least significant 8 bits of the initial 10 bit frequency data
 byte 2: MSN = 4 bit ATN data (constant throughout the note)
 LSN = 0 0 F0 F1, the top 2 bits of the initial frequency data
 byte 3: NLEN, number of steps in the frequency sweep, 1 to 255 (0 => 256)
 byte 4: FPS : FPSV, prescaler reload value and initial FPSV
 byte 5: FSTEP, sweep step size, 1 to 127, -1 to -128

* Type 2: Fixed frequency, swept attenuation

See Figure 6.

byte 0: header, CH# in B7 - B6, note type = 2 in B1 - B0
 byte 1: least significant 8 bits of the 10 bit frequency data (constant)

byte 2: MSN = 4 bit ATN data (initial value)
 LSN = 0 0 F0 F1, the top 2 bits of the frequency data (constant)
 byte 3: NLEN, duration of the note = NLEN * 16.7ms
 byte 4: ALEN : ASTEP
 ALEN = number of steps in the attenuation sweep
 ASTEP = step size, 1 to 7, -1 to -8
 byte 5: APS : APSV, prescaler reload value and initial APSV, 1 to 15 (0 => 16)

* Type 3: Swept frequency, swept attenuation

See Figure 7.

byte 0: header, CH# in B7 - B6, note type = 3 in B1 - B0
 byte 1: least significant 8 bits of the initial 10 bit frequency data
 byte 2: MSN = 4 bit ATN data (initial value)
 LSN = 0 0 F0 F1, the top 2 bits of the initial frequency data
 byte 3: NLEN, number of steps in the frequency sweep, 2 to 255 (0 => 256)
 byte 4: FPS : FPSV, prescaler reload value and initial FPSV, 0 - 15 (0 => 16)
 byte 5: FSTEP, sweep step size, 1 to 127, -1 to -128
 byte 6: ALEN : ASTEP
 ALEN = number of steps in the attenuation sweep
 ASTEP = step size, 1 to 7, -1 to -8
 byte 7: APS : APSV, prescaler reload value and initial APSV, 1 to 15 (0 => 16)

* Noise notes: special case Type 2 notes

See Figure 8.

Noise notes are notes that are played on the sound chip noise generator (CH#0). They are stored in ROM as a special case of a Type 2 note, fixed frequency and swept attenuation. They consist of white noise with superimposed attenuation decay, which creates a percussive effect, such as a snare drum note.

Instead of frequency information, noise notes are stored with three bits of noise control data: FB NF0 NF1 (see TI data sheets 76489), which remain constant throughout the note. Experimentation with various values can result in credible percussion effects.

NLEN, as is the case for a regular Type 2 note, directly determines the duration of a noise note.

The sound chip noise generator is unlike the other generators in that sending it redundant data (i.e., the same data that it has stored in its internal registers) has an audible effect on its output. In particular, whenever control data, redundant or not, is sent to the noise generator, its internal shift register is reset, causing a short pop or click to be heard. This isn't annoying on an occasional basis, or when a new noise starts, but remember: PLAY_SONGS is sending data to all four channels every 16.7ms. This would cause a noticeable lack of "whiteness" in the noise generator's output.

PLAY_SONGS avoids this problem by referencing a dedicated CART RAM location, SAVE_CTRL (see Figure 9) each time before it sends control data to the noise generator. SAVE_CTRL contains the control data that was output to the noise generator the last time through PLAY_SONGS. If the noise control data in the pointed to song data area = SAVE_CTRL, PLAY_SONGS doesn't send it out again. If there is new data to be sent, that data is output and SAVE_CTRL is updated.

byte 0: header, CH#0 in B7 - B6, note type = 2 in B1 - B0
byte 1: MSN = 4 bit ATN data (initial value)
LSN = 0 FB NF0 NF1, noise control data
byte 2: NLEN, duration of note: $NLEN * 16.7ms$
byte 3: ALEN : ASTEP
ALEN = number of steps in the attenuation sweep
ASTEP = step size, 1 to 7, -1 to -8
byte 4: APS : APSV, prescaler reload value and initial APSV, 1 to 15 (0 => 16)

OPERATING SYSTEM ROUTINES

INIT_SOUND

Contains ENTRY POINT: ALL_OFF

INIT_SOUND, usually called right after power on, turns off the sound generators, initializes the CART RAM locations to be used as song data areas, and sets up the four channel data area pointers. Specifically, it:

- 1) directly turns off all four sound generators.
- 2) initializes PTR_TO_LST_OF_SND_ADDRS, a dedicated 16 bit CPU RAM pointer which other sound routines expect to contain the base address of a list in CART ROM (called LST_OF_SND_ADDRS) of the starting addresses of each song's data area and note list. The address of LST_OF_SND_ADDRS is passed to INIT_SOUND in HL.
- 3) stores the sound-inactive code (OFFH) into byte 0 of n song data areas. n is passed in B and = the total number of song data areas used by the game.
- 4) stores an end of data area code (00) following the last data area.
- 5) sets the four pointers to the data areas for the songs to be played on each channel, PTR_TO_S_ON_x (x = 0-3), to a dummy inactive area (DUM_AREA, which is actually a single OFFH byte within INIT_SOU).
- 6) sets SAVE_CTRL to an initial value of OFFH

INPUT: n
TYPE: 8 bit constant
PASSED: in B
DESCRIPTION: number of song data area used by the game

INPUT: LST_OF_SND_ADDRS
TYPE: 16 bit address
PASSED: in HL
DESCRIPTION: LST_OF_SND_ADDRS is the base address of a list of the starting addresses of each song's data area and note list.

OUTPUT: 1) turns off all sound generators
2) initializes PTR_TO_LST_OF_SND_ADDRS
3) writes inactive code to byte 0 of n song data areas
4) stores 00 at end of song data areas
5) sets the 4 channel song pointers to the inactive DUM_AREA
6) sets SAVE_CTRL to OFFH

ALL_OFF

ALL_OFF directly turns off all four sound generators, but does nothing to any song data areas or the 4 channel data pointers.

INPUT: none

OUTPUT: turns off all sound generators

JUKE_BOX

JUKE_BOX is called to start a song. Using a song number passed in B, JUKE_BOX loads the data for the song's first note into the appropriate song data area (the address of the area is found by using the song number as an index into the LST_OF_SND_ADDRS table). It also formats the data area's header and sets up the next note pointer. If the song is a special sound effect, its next note pointer is set to the address of the special effect routine. The next time PLAY_SONGS is called, that song's first note will be processed (thereby truncating whatever song had been "playing" in that data area), and the song will have started.

Since starting a new song may have altered the priority structure within the song data areas, JUKE_BOX also calls UP_CH_DATA_PTRS to modify the channel data pointers accordingly.

If JUKE_BOX is called with a song number of a song already in progress, it returns immediately (i.e., it doesn't restart the song).

INPUT: song number to be started
TYPE: 8 bit constant, 1 to 61
PASSED: in B

CALLS: PT_IX_TO_SxDATA, LOAD_NEXT_NOTE, UP_CH_DATA_PTRS

OUTPUT: 1) moves the song's first note data to the appropriate song data area
1) formats byte 0 header of the song's data area
2) points next note pointer in data area (bytes 1&2) to address of first note in song, or address of special sound effect routine
3) updates the channel data pointers on basis of song priorities

SND_MANAGER

SND_MANAGER should be called every VDP interrupt (every 16.7 ms). It assumes that the song data areas are stored contiguously in a data block beginning with the data area assigned to song number one. For each data area, SND_MANAGER, or routines which it calls, processes the appropriate timer and sweep counters and modifies the frequency and attenuation data accordingly. If the data area is assigned to a special effect, SND_MANAGER simply calls that effect, and doesn't modify any data. When a note is finished, SND_MANAGER, using the data area's next note pointer, moves data for the next note of the song into the area and fills in keys bytes within the area to allow proper processing of the data area by the sweep routines it calls (FREQ_SWEEP and ATN_SWEEP). (SND_MANAGER considers a note finished when its frequency duration timers have timed out; see the descriptions of the FREQ_SWEEP and ATN_SWEEP routines) A special effect is responsible for deciding when its over and initiating the next note in the song.

After the operations upon a data area have been performed, the channel data area pointers (PTR_TO_S_ON_x) may be updated (see description of UP_CH_DATA_PTRS in "UTILITIES" section).

If SND_MANAGER reads a header byte (in CART ROM) that has bits 3&4 set, indicating repeat song, it will start the song again by reloading the first note in the song, using the SONGNO portion (B5-B0) of byte 0 in the song's data and the LST_OF_SND_ADDRS to find it.

SND_MANAGER does not output the modified frequency and attenuation data. PLAY_SONGS is usually called just before SND_MANAGER to do this.

Special codes in byte 0 of the song data area indicate:

- | | | |
|------------|---|---|
| 0FFH | - | data area inactive, do no processing, do not modify channel data area pointer |
| B5-B0 = 62 | - | a special effect is to be played; SND_MANAGER calls the effect routine |
| 00H | - | end of song data areas (SND_MANAGER processes data areas until it sees 0 in byte 0) |

NOTE: Song number 1 MUST use the first data area in the block of song data areas.

INPUT: none

CALLS: PROCESS_DATA_AREA, PT_IX_TO_SxDATA

OUTPUT: 1) decrements song duration and sweep timers
2) modifies swept frequency and attenuation values
3) calls special effects routines where necessary
4) restarts the song if indicated
5) may update the channel data area pointers (PTR_TO_S_ON_x)

PLAYSONGS

PLAY_SONGS takes the frequency and attenuation data pointed to by the four channel data area pointers (**PTR_TO_S_ON_x**) and outputs it to the four sound chip generators. Action is taken on the basis of the each data area's byte 0:

- 1) If the pointed to data area is active, the frequency and attenuation data are sent to the channel indicated by B7-B6 (**CH#**) of byte 0 of the pointed to data area.
- 2) If byte 0 is OFFH (inactive), the channel to which that pointer is dedicated is sent the OFF attenuation code.
- 3) If **CH#** = 0 (noise), the attenuation data is output. If there is no new noise control data to be output (determined by checking dedicated CART RAM location **SAVE_CTRL**), no control data is sent out. Otherwise, the new control data is output and **SAVE_CTRL** is updated.

INPUT: none

OUTPUT: through **SOUND_PORT**,

- 1) current freq and atn data is output to each tone generator, if song/effect on that channel is active
- 2) noise generator is sent current atn data, and control data, if new
- 3) modifies **SAVE_CTRL** if necessary

FREQ_SWEEP

FREQ_SWEEP is used by **SND_MANAGER** and special effects routines to create frequency sweeps. It operates upon frequency data stored within a song data area, and is normally called (by **SND_MANAGER** or a special effect routine) once every VDP interrupt (16.7ms). The start of the data area (address of byte 0) is passed in IX.

FREQ_SWEEP assumes data has been stored as follows (names which may be used to describe the various bytes or byte segments within the data area are indicated; see Figure 1):

- byte 3: the least significant 8 bits of that note's frequency ($F_2 - F_9$)
- byte 4: top 2 bits of that note's frequency: $B_1 = F_0$, $B_2 = F_1$
- byte 5: **NLEN** - determines the note's duration:
 - 1) if frequency is to be swept, **NLEN** = number of steps in the sweep: 2 to 255 (0 => 256)
 - 2) if fixed frequency, $\text{NLEN} * 16.7 \text{ ms} = \text{duration of the note}$: 1 to 255 (0 => 256)
- byte 6: **FPS** : **FPSV** - frequency sweep duration prescaler:
 - FPS** = prescaler reload value: 0 to 15 (0 => 16)
 - FPSV** = temp storage nibble for **FPS**: init ROM value, 0 to 15 (0 => 16)
 - duration of sweep (& note) = $[(\text{NLEN}-1) * \text{FPS}] + \text{initial FPSV} * 16.7\text{ms}$
 - duration 1st step = $\text{initial FPSV} * 16.7\text{ms}$
 - duration all other steps = $\text{FPS} * 16.7\text{ms}$
- byte 7: **FSTEP** - frequency sweep step size: signed 8 bit number, two's complement: 1 to 127, -1 to -128
 - if **FSTEP** = 00, frequency is not to be swept, but **NLEN** is decremented each time called

Parameter limitations:

- 1) In a frequency sweep, a "step" consists of a single fixed frequency tone; therefore, the minimum number of steps a frequency sweep can have is two (otherwise the frequency wouldn't have "swept").
- 2) If a note is to be frequency swept, **FSTEP** must not be 0.
- 3) The minimum length fixed frequency note has **NLEN** = 1.
- 4) Maximum **NLEN** 0, which is equivalent to 256.

FREQ_SWEEP returns with the Z flag SET if the note (swept or fixed) is over, RESET if the note is not over. (**PROCESS_DATA_AREA** decides that a note is over when **FREQ_SWEEP** returns with the Z flag set)

INPUT: 16 bit address of a song data area in CPU RAM

PASSED: in IX

DESCRIPTION: **FREQ_SWEEP** operates upon frequency data within this song data area

- OUTPUT:**
- 1) duration and sweep counters are decremented
 - 2) freq data in bytes 3&4 is modified if note is freq swept
 - 3) returns with Z flag SET if note over, RESET if note not over

ATN_SWEEP

ATN_SWEEP is used to create attenuation sweeps. It operates upon attenuation data stored within a song data area, and is normally called (by PROCESS_DATA_AREA or a special effect routine) once every VDP interrupt (16.7ms). The start of the data area (address of byte 0) is passed in IX.

ATN_SWEEP assumes data has been stored as follows (see Figure 1):

byte 4: ATN - the MSN = 4 bit attenuation

byte 8: ALEN : ASTEP - no-sweep code or sweep length and step size:
 1) if byte 8 = 00, ATN is not to be swept and counters aren't changed
 2) if byte 8 non zero, attenuation is to be swept:
 a) ALEN = number of steps in the sweep: 1 to 15 (0 => 16)
 b) ASTEP = sweep step size: 1 to 7, -1 to -8 (signed, 4 bit two's complement)

byte 9: APS : APSV - attenuation sweep duration prescaler:
 1) if attenuation is not swept, byte 9 is not used by ATN_SWEEP
 2) if attenuation is to be swept:
 APS = prescaler reload value: 1 to 15 (0 => 16)
 APSV = temp storage nibble for APS: init ROM value, 1 to 15 (0 => 16)
 duration of swept attenuation part of note =
 $[(ALEN - 1) * APS] + \text{initial APSV} * 16.7 \text{ ms}$
 duration 1st step = initial APSV * 16.7ms
 duration all other steps = APS * 16.7ms

Parameter limitations:

- 1) In an attenuation sweep, a "step" consists of a tone (swept or not) played at a fixed attenuation level; so, the minimum number of steps an attenuation sweep can have is two (otherwise the attenuation wouldn't have "swept"). Therefore, the minimum ALEN value is 2 (0 => 16)
- 2) If a note is to be attenuation swept, byte 8 must not be 00.
- 3) The absolute value of ASTEP must be >= 1.

If byte 8 is 00, ATN_SWEEP returns immediately with Z flag SET (the sweep is over or the note was never swept), and doesn't modify any counters. When a sweep finishes, ATN_SWEEP sets byte 8 to 00 and returns with the Z flag SET. If a sweep is in progress, ATN_SWEEP returns with the Z flag RESET. (NOTE: PROCESS_DATA_AREA decides that a note is over when FREQ_SWEEP returns with Z set: the length of a note has nothing to do with when its attn sweep is over)

INPUT: 16 bit address of a song data area in CPU RAM

PASSED: in IX

DESCRIPTION: ATN_SWEEP operates upon frequency data within this song data area

OUTPUT: 1) duration and sweep counters are decremented if sweep in progress
 2) atn data in byte 4 is modified if note is atn swept
 3) RETs C SET, byte 8 = 0 if sweep is over or note was never swept
 RETs C RESET if sweep in progress

PROCESS_DATA_AREA

PROCESS_DATA_AREA is called by **SND_MANAGER**. For an active data area (address of byte 0 passed in **IX**), **PROCESS_DATA_AREA** modifies the timers, sweep counters, frequency, and attenuation data by calling **FREQ_SWEEP** and **ATN_SWEEP**. If a note finishes during the current pass through **PROCESS_DATA_AREA**, the next note in the song is examined and its data is loaded into the data area (calls **LOAD_NEXT_NOTE**). Then, in order to maintain the song data area priority structure, the **CH# : SONGNO** of the newly loaded note is compared to the **CH# : SONGNO** of the previous note: if there is a difference, **UP_CH_DATA_PTRS** is called to adjust the channel data area pointers in response to the change caused by loading the next note.

If the data area is being used by a special sound effect, **PROCESS_DATA_AREA** calls the sound effect routine whose address is stored in bytes 1&2 of the data area (the actual address called is routine + 7: see discussion of special sound effects).

If the data area is inactive, **PROCESS_DATA_AREA** returns immediately (no processing occurs).

INPUT: address of byte 0 of a song data area
PASSED: in **IX**

CALLS: **ATN_SWEEP**, **FREQ_SWEEP**, **LOAD_NEXT_NOTE**, **UP_CH_DATA_PTRS**, **AREA_SONG_IS**

OUTPUT: 1) if active, modifies song data area's timer, freq, and atn data
2) loads the next note's data when a note is finished
3) if special sound effect routine using data area, calls it
4) when necessary, updates the channel data area pointers

LOAD_NEXT_NOTE

Called by **PROCESS_DATA_AREA** and **JUKE_BOX**, **LOAD_NEXT_NOTE** examines the next note to be played in a data area (address byte 0 passed in IX) and moves its data into the area. It fills in bytes (e.g., to indicate swept or not swept) where appropriate, based upon note type. If the next "note" is a special sound effect, its address is saved in bytes 1&2 and the address of the routine + 0 is called, with the address of the note to follow the effect passed in HL and **SONGNO** passed in A. This will cause the special effect routine to save both these values. Then, the special effect routine + 7 is called, which allows the routine to initialize the song data area for the first pass through **PLAY_SONGS**. (see discussion of special sound effects)

Prior to moving the next note data, **LOAD_NEXT_NOTE** saves the data area's byte 0 (**CH# : SONGNO**) and stores the song inactive code (**OFFH**) there. The last thing **LOAD_NEXT_NOTE** does is restore byte 0, loading **CH#** with the **CH# : SONGNO** of the new note (usually the same as the old note). If the new note is a special sound effect, 62 is returned as the **SONGNO** part of byte 0.

INPUT: address of byte 0 of a song data area
PASSED: in IX

OUTPUT:

- 1) sets up song data area with data from next note to be played
- 2) for next note = special sound effect, calls the effect twice, first with the address of the following note in the song and the song's **SONGNO**, and then once more to allow the effect to initialize the song data area
- 3) if next note is "normal", loads **CH# : SONGNO** in byte 0 with **CH# : SONGNO** of new note
- 4) returns with byte 0 = **OFFH** if song over, **SONGNO** = 62 if next note is a sound effect

UTILITIES

The following are O/S utility routines, used by the main O/S sound programs, that may be of use to the cartridge programmer:

*** AREA_SONG_IS ***

The address of byte 0 of a song data area is passed in IX. The song number of the song using that area is returned in A (OFFH if inactive). If a special effect was using that area, 62 is returned in A and HL is returned with the address of the special sound effect routine.

*** UPATNCTRL ***

Perform single byte update of the snd chip noise control register or any attenuation register. IX is passed pointing to byte 0 of a song data area. MSN register C = formatted channel attenuation code.

*** UPFREQ ***

Perform double byte update of a sound chip frequency register. IX is passed pointing to byte 0 of a song data area. MSN register D = formatted channel frequency code.

*** DECLSN ***

Without affecting the MSN, decrement the LSN of the byte pointed to by HL. HL remains the same.

RET with Z flag set if dec LSN results in 0, reset otherwise.

RET with C flag set if dec LSN results in -1, reset otherwise.

*** DECMSN ***

Without affecting the LSN, decrement the MSN of the byte pointed to by HL. HL remains the same.

RET with Z flag set if dec MSN results in 0, reset otherwise.

RET with C flag set if dec MSN results in -1, reset otherwise.

*** MSNTOLSN ***

Copy MSN of the byte pointed to by HL to the LSN of that byte. HL remains the same.

*** ADD816 ***

Adds 8 bit two's complement signed value passed in A to the 16 bit location pointed to by HL. Result is stored in the 16 bit location.

*** PT_IX_TO_SxDATA ***

A SONGNO is passed in B. PT_IX_TO_SxDATA returns with IX pointing to the song data area which is used by that SONGNO.

*** UP_CH_DATA_PTRS ***

UP_CH_DATA_PTRS adjusts each channel data pointer to point to the highest priority (ordinal last) song data area that uses that channel. It is called whenever a change has been made to a song data area that requires modification of the channel data pointers.

All 4 channel data pointers (PTR_TO_S_ON_x) are initially pointed to a dummy inactive area, DUM_AREA. Then, moving in order from the first data area to the last, CH# in byte 0 of each data area is examined, and the corresponding channel data pointer is pointed to that data area. Thus, by the time the routine is done, each channel data pointer is pointing to the last active data area that contains data to be sent to that channel. If none of the active data areas used a particular channel, then that channel remains pointing to DUM_AREA (and therefore its generator will be turned off next time through PLAY_SONGS).

*** LEAVE_EFFECT ***

LEAVE_EFFECT, called by a special sound effect routine when it's finished, restores the SONGNO of the song to which the effect belongs to B5 - B0 of byte 0 in the effect's data area, and loads bytes 1 & 2 with the address of the next note in the song. The address of the 1 byte SONGNO (saved by the effect when it was first called) is passed in DE. The 2 byte address of the next note in the song, also saved by the effect, is passed in HL. IX is assumed to be pointing to byte 0 of the data area to which the song number is to be restored. Bits 7 & 6 of the saved SONGNO byte are not stored into byte 0, and therefore may be used during the course of the effect to store any useful flag information.

SPECIAL SOUND EFFECTS

* Sound effects as notes within a song

Sounds which do not fit one of the six categories of "normal" musical notes can be created and played throughout the course of a song as "special effect" notes. Unlike normal musical notes, which are stored in ROM as tables of frequency/control and attenuation data, a special effect's data are determined algorithmically by a custom routine written by the cartridge programmer. Special effect notes can also be used to generate sounds that could have been comprised of many normal notes, but which are more efficiently (in terms of ROM space used) computed by a short program.

These notes use the same song data area as the song within which they are contained, and they are stored in the song's ROM note list with a one byte header as are normal notes. However, the bytes following the ROM header do not contain data to be directly loaded into the song data area. The header (see Figure 2), which specifies the channel upon which to play the effect (which is usually the same as the channel used by the rest of the notes in the song), is followed by a two byte address of a routine written by the cartridge programmer which will be called every 16.7ms by PROCESS_DATA_AREA. When called, this special effect routine should compute data values and store them at the appropriate locations within the song data area. (In fact, many effect routines may call the O/S routines FREQ_SWEEP or ATN_SWEEP, which also require that data be ordered appropriately within a song data area) This computed data will then be output on the next pass through PLAY_SONGS (assuming that this song data area has the highest priority of any data area using the same channel).

Variables required by the effect which will not be output may be stored wherever the programmer desires. Free locations within the song's data area might as well be used for effect variable storage, since the entire ten byte area is reserved for the song anyway. If no free locations exist within a data area, which would be the case if an effect required both frequency and attenuation to be swept, the effect can store the remaining needed variables wherever convenient.

In order to interact properly with the O/S sound routines, each special effect routine must conform to a certain format. A description of that format, and how an effect interacts with the O/S routines, follows:

WHEN AN EFFECT BEGINS - When loading a new note, if `LOAD_NEXT_NOTE` sees that the note to be loaded is a special effect:

1) It stores in byte 0 of the song's data area the effect's `CH#` and a `SONGNO` of 62. `SONGNO = 62` is used later by `PROCESS_DATA_AREA` to detect the fact that an effect is using the data area.

2) It then takes the address of the special effect routine (lets's call it `SFX`) from ROM and puts it into bytes 1&2 (`NEXT_NOTE_PTR`).

3) `LOAD_NEXT_NOTE` then calculates the ROM address of the header of the next note in the song, stores that address in `HL`, puts the song's `SONGNO` in `A`, and calls `SFX + 0`. In every special effect routine at `SFX + 0`, there **MUST** be the following code which saves the two passed values (see Figure 9):

```
SFX:      LD (SAVE_x_NNP),HL
          LD (SAVE_x_SONGNO),A
          RET
SFX+7:    code for sound effect starts here
          ...
```

where `SAVE_x_NNP` is a two byte location used by all the sound effect notes in the current song to save the address of the next note in the song, and `SAVE_x_SONGNO` is the address of a byte where the song number is saved. The programmer may put `SAVE_x_NNP` and `SAVE_x_SONGNO` wherever desired, including somewhere within the song data area.

Thus, calling `SFX + 0` allows each effect routine to save the next note's address and the song's `SONGNO`.

1ST PASS THROUGH EFFECT - After calling `SFX + 0`, `LOAD_NEXT_NOTE` calls `SFX + 7` for the first pass through the body of the routine. At this location, there should be code which initializes the appropriate bytes within the song data area, as the next pass through `PLAY_SONGS`, subject to the data area priority system, will output this initial data in normal fashion.

As will be seen below, this same location (`SFX + 7`) will be called every 16.7ms by `PROCESS_DATA_AREA` to modify the data within the area. Therefore, the code at `SFX + 7` must know which pass is in effect, so that the song data area will be initialized only on the first pass. A convenient way of doing this is to test bit 7 of `SAVE_x_SONGNO`, the byte which contains the saved song number. On the 1st pass through the effect, bit 7 (and bit 6) will be zero, since the largest possible `SONGNO` (62) would not set this bit. If bit 7 is zero, then, code to initialize the data area can be executed and bit 7 reset to prevent re-initialization. I.e.,

```

SFX+7:  LD HL,SAVE_x_SONGNO
        BIT 7,(HL)
        JR NZ,NOT_PASS_1
        SET 7,(HL)      ;to prevent further passes thru inits
        ...             ;initialize bytes within the data area here
        RET             ;to LOAD_NEXT_NOTE
NOT_PASS_1: ....        ;code for pass 2 or greater starts here

```

PRIORITY UPDATE - After calling SFX + 0 and SFX + 7, LOAD_NEXT_NOTE will return to PROCESS_DATA_AREA, which checks to see if loading a new note has caused a change in either the channel used by the song (this happens with noise notes within a musical song) or the song number. If a change has occurred, UP_CH_DATA_PTRS will be called, which updates the data pointers on the basis of priority within the block of song data areas (see description of this routine in the preceeding "UTILITIES" section). Since a special effect note will cause a change in the song number (from whatever it was to 62), UP_CH_DATA_PTRS will always be called whenever an effect note is loaded.

SECOND PASS OR GREATER - The next time PROCESS_DATA_AREA is called (from SND_MANAGER), which will be 16.7ms after PLAY_SONGS has sent out the effect's initial data, it will detect the fact that an effect is using the song data area (by seeing a SONGNO of 62) and will JUMP to SFX + 7, rather than calling the frequency and attenuation sweep routines as it would for a normal note. This will result in the first pass through the part of the body of the effect routine that actually does computation and adjusts the data values within the data area. When the effect routine has completed its modifications to the data area and performs a RET, control is transferred back to SND_MANAGER, which then moves on to the next song data area to be processed.

This process will be repeated every 16.7ms until the effect routine itself decides that it's over and takes action to load the next note in the song.

WHEN AN EFFECT IS OVER - Prior to performing a RET, the effect routine must decide whether the effect note has finished. If it has, NEXT_NOTE_PTR within the data area must be set to the address of the next note in the song and SONGNO must be restored to byte 0. This can be done by calling the O/S routine LEAVE_EFFECT which does this. The address of SAVE_x_NNP must be passed in HL and the address of SAVE_x_SONGNO must be passed in DE. Finally, the effect should JUMP to EFXOVER, a location within PROCESS_DATA_AREA which would normally be reached once a note is over. The code there takes care of loading the next note in the song. Thus, the final code of each effect routine will look as follows:

```

RET if effect not over
LD HL,(SAVE_x_NNP)      ;HL := addr next note in song
LD DE,SAVE_x_SONGNO     ;DE := addr saved song number
CALL LEAVE_EFFECT       ;to restore them to bytes 0 - 2 in data area
JP EFXOVER              ;in PROCESS_DATA_AREA to load song's next note

```

The entire above described sequence is summarized in Figure 9.

* A sound effect as a single sound

A stand alone sound effect can be implemented within the previously mentioned structures simply by creating a single note song. The single note is the effect

and would be followed by an end of song code (or repeat code if you wish the effect to go on forever).

Many stand alone effects may want to use more than one tone generator channel: e.g., a special laser zap that momentarily requires all three tone generators, or, as is often the case, a white noise effect of particular character that requires the noise channel shift rate to be modified by channel three (see TI data sheets). In these cases, the effect's routine will have to modify data in several data areas whenever called. The song data areas used by such effects are subject to the normal priority structure. E.g., if you wish a two channel effect to temporarily overwrite the harmony and bass lines of a repeating song, the effect must have been assigned two data areas of higher priority (ordinally later in the block of song data areas). If it is not necessary to maintain any underlying songs, an effect can share data areas to conserve RAM space, with the understanding that, as usual, songs or sounds that share the same song data area truncate each other. A multi-channel effect (a chord note, say) may be included as a note within a song, but, again, the song data area priority structure determines what will finally be heard.

Providing for a typical game's sound generation needs might require eight song data areas: four for an underlying, repeating song(s) (three areas for the three tone generators and one for the noise generator used for percussion notes), and four for higher priority, occasional sound effects (which would temporarily overwrite the repeating songs, but truncate each other).

* Pseudo code listings of main routines

The following two pages contain pseudo code descriptions of most of the O/S sound routines. Some computational details are not shown, but all jumps, calls, returns, pushes and pops are listed.

Terminology:

"=" is used as the assignment statement, and "(xx)" means the contents of the memory location pointed to by xx, where "xx" is HL, IX, etc.

The structure of each description is as follows:

*** name of routine ***

the value expected for passed parameters (if any)

pseudo code description
of the routine, uninterrupted
by blank lines
RET

```

*** JUKE_BOX ***
B = SONGNO to be started

PUSH BC
CALL PT_IX_TO_S:DATA (IX set)
POP BC
RET if song in progress
    byte 0 := SONGNO
    set NEXT_NOTE_PTR to 1st note in song
    CALL LOAD_NEXT_NOTE
    CALL UP_CH_DATA_PTRS
    RET

*** PLAY_SONGS ***

set A for CH1 OFF code
set MSN C for CH1 attenuation
set MSN D for CH1 frequency
IX := (PTR_TO_S_ON_1)
(i.e., pt IX to byte 0 data area of song for CH1)
CALL TONE_OUT
set A for CH2 OFF code
set MSN C for CH2 attenuation
set MSN D for CH2 frequency
IX := (PTR_TO_S_ON_2)
(i.e., pt IX to byte 0 data area of song for CH2)
CALL TONE_OUT
set A for CH3 OFF code
set MSN C for CH3 attenuation
set MSN D for CH3 frequency
IX := (PTR_TO_S_ON_3)
(i.e., pt IX to byte 0 data area of song for CH3)
CALL TONE_OUT
set A for CH0 OFF code
set MSN C for CH0 attenuation
IX := (PTR_TO_S_ON_0)
(i.e., pt IX to byte 0 data area of song for CH0)
IF area INACTIVE
    turn off CH0
ELSE
    CALL UPATNCTRL (send current atn)
    set LSN A for current ctrl data
    IF current ctrl data diff from last
        reload SAVE_CTRL
    CALL UPATNCTRL (send new ctrl)
ENDIF
ENDIF
RET

*** SND_MANAGER ***

CALL PT_IX_TO_S:DATA, song 01
LOOP
    RET if end of song data areas
    CALL PROCESS_DATA_AREA
    pt IX to byte 0 next song data area
REPEAT LOOP

*** PROCESS_DATA_AREA ***
IX = addr byte 0 of a song data area

CALL AREA_SONG_IS
RET if area INACTIVE
    IF SONGNO = 62
        JP SFX+7 (RET from SFX)
    ENDIF
    CALL ATM_SWEEP
    CALL FREQ_SWEEP
    IF note over
EFXOVER: PUSH CH0 : SONGNO note just over
    CALL LOAD_NEXT_NOTE
    POP CH0 : SONGNO note just over
    IF CH0 : SONGNO newly loaded note not =
    CH0 : SONGNO note just over
        CALL UP_CH_DATA_PTRS
    ENDIF
ENDIF
RET

```

```

*** LOAD_NEXT_NOTE ***
IX = addr byte 0 of a song data area
byte 0 = CH0 (or 00) : SONGNO
(SFX = addr of a special effect note's routine)

PUSH 0 0 : SONGNO (CH0 not pushed)
deactivate area (byte 0 := FF)
A := (NEXT_NOTE_PTR) (header new ROM note)

CASE header type of
1)rest:
    PUSH header of new ROM note
    set NEXT_NOTE_PTR for next note in song
    (i.e., the note after this new note)
    set bytes in song data area:
        ATM := off
        MLEN := 5 bit rest duration
        FSTEP := 0 (i.e., no freq sweep)
        ASTEP := 0 (no atn sweep)
    JP MODB0
2)end of song:
    IF end repeat
        POP BC (B := SONGNO)
        CALL JUKE_BOX to reload 1st note of this song
        RET (to PROCESS_DATA_AREA)
    ENDIF
    PUSH inactive code
    JP MODB0
3)special effect:
    POP IY (IY := SONGNO)
    PUSH IY to put SONGNO back on stack
    PUSH header new ROM note
    set NEXT_NOTE_PTR, bytes 1&2, to SFX
    (address special effect routine)
    set DE to SFX
    HL := addr next note in song
    (i.e., the note after this new effect note)
    PUSH IY, POP AF (A := SONGNO)
    PUSH DE, POP IY (IY := SFX)
    DE := PASS1, PUSH DE
    JP (IY), i.e., "CALL (IY)", RET to PASS1
    (SFX saves SONGNO & addr next note)
    PASS1: IY := IY + 7
    DE := MODB0, PUSH DE
    JP (IY), i.e., "CALL (IY+7)", RET to MODB0
    (SFX+7 loads initial effect data)
4)normal note:
CASE note type
    0)NEXT_NOTE_PTR := adr song's next note
    move 3 bytes ROM note data to RAM
    FSTEP := 0 (no freq sweep)
    ASTEP := 0 (no atn sweep)
    JR MODB0
    1)NEXT_NOTE_PTR := adr song's next note
    move 5 bytes ROM note data to RAM
    ASTEP := 0 (no atn sweep)
    2)NEXT_NOTE_PTR := adr song's next note
    move 5 bytes ROM note data to RAM
    FSTEP := 0 (no freq sweep)
    JR MODB0
    3)NEXT_NOTE_PTR := adr song's next note
    move 7 bytes ROM note data to RAM
ENDCASE
ENDCASE

MODB0: PUSH IX
POP HL to point to byte 0
POP AF (A := header new note)
POP BC (B := SONGNO)
RET if header is inactive (i.e., song is over)
    IF header is for a special effect
        B := 62, the SONGNO for all effect notes
    ENDIF
    byte 0 := CH0 (from header new note) : SONGNO (from B)
    RET

```

*** INIT_SOUND ***

HL = addr of LST_OF_SND_ADDRS
B = number of song data areas used by game

set RAM word PTR_TO_LST_OF_SND_ADDRS to value passed in HL
pt HL to byte 0 in 1st song data area
B1 (HL) := inactive code (OFFH)
HL := HL + 10
DJNZ B1 (set B areas inactive)
(HL) := end of data area code (0)
load all 4 channel data area pointers with the
addr of a dummy inactive area (DUMAREA)
SAVE_CTRL := OFFH
ALL_OFF: turn off all 4 generators directly
RET

DUMAREA DEFB OFFH

*** UP_CH_DATA_PTRS ***

PUSH IX to save it
set all 4 ch data ptrs to a dummy, inactive area
CALL PT_IX_TO_SxDATA, song 0 1
LOOP
IF byte 0 indicates the end of the song data areas JR DONE
IF byte 0 indicates an active area
set HL to address of this area's channel data pointer
(i.e., HL := addr PTR_TO_S_ON_0 + (CH# this area * 2))
PUSH IX
POP DE (DE := addr byte 0 this area)
(HL) := E, (HL+1) := D
ENDIF
IX := IX + 10
ENDIF
REPEAT LOOP
DONE POP IX to restore it
RET

*** TONE_OUT ***

IX = (PTR_TO_S_ON_0), i.e., IX pts to byte 0 data area of song for CH#
A set for CH# OFF code
MSN C set for CH# attenuation
MSN D set for CH# frequency

IF area INACTIVE
turn off CH#
ELSE
CALL UPATMCTRL (send out attenuation)
CALL UPFREQ (send out frequency)
ENDIF
RET

*** PT_IX_TO_SxDATA ***

B = a song number

HL := addr of LST_OF_SND_ADDRS
HL := HL - 2
BC := 4 * SONGNO
HL := HL + BC (i.e., HL now points to SxDATA's entry in LST_OF_SND_ADDRS)
E := (HL), D := (HL+1)
PUSH DE
POP IX (IX := addr byte 0 of this song's data area)
RET

*** FREQ_SWEEP ***

IX = Addr byte 0 of a song data area

IF FSTEP = 0 note is not to be swept
A := NLEN
DEC A
RET Z (leave if note over, Z flag SET)
NLEN := A (store decremented NLEN)
RET (note not over, Z flag RESET)
ENDIF
PUSH IX, POP HL (pt HL to byte 0)
HL := HL + offset within data area of FPSV
CALL DECSN to decrement FPSV
IF Z flag SET, FPSV has timed out
CALL MSNTOLN to reload FPSV
A := NLEN
DEC A
RET Z (leave if sweep over with Z flag SET)
NLEN := A (store decremented NLEN)
point HL to FREQ
A := FSTEP
CALL ADDB16 to add FSTEP to FREQ
RESET bit Z in hl byte FREQ
(in case of overflow from addition)
OR OFFH to RESET Z flag
ENDIF
RET

*** ATH_SWEEP ***

IX = addr byte 0 of a song data area

RET with Z flag SET if byte B = 0
(i.e., note ath not to be swept)
PUSH IX, POP HL (pt HL to byte 0)
HL := HL + offset within data area of APSV
CALL DECSN to decrement APSV
IF Z flag SET, APSV has timed out
CALL MSNTOLN to reload APSV
pt HL to ALEN (0 of steps in ath sweep)
CALL DECSN to decrement ALEN
IF Z flag RESET, sweep not over yet
ATH := ATH + ASTEP
(4 bit add, overflow ignored)
OR OFFH to RESET Z flag
ELSE Z flag is SET (sweep is over)
byte B := 0 to indicate sweep over
ENDIF
ENDIF
RET

SxDATA

FIGURE 1

Description: Storage area for the various timers and output data for song number x. The song data areas MUST be stored in a contiguous block of CPU RAM and the data area used by song number one MUST be the first data area in the block. Song data area storage is allocated according to addresses stored in LST_OF_SND_ADDRS, a table stored in CART ROM.

Byte 0 of each data area, in addition to giving CH# and song number, can indicate two special conditions:

- byte 0 = OFFH: song(s) using this data area are inactive
- byte 0 = 00H: indicates end of song data areas

If SONGNO = 62, the address of a special sound effect routine is stored in bytes 1 and 2.

Length in bytes: 10

Location: CPU RAM

Beginning address: pointed to by a 16 bit entry in LST_OF_SND_ADDRS

Offset	Contents								Description
	B7	B6	B5	B4	B3	B2	B1	B0	
0	CH#		SONGNO				B7 - B6: song channel number, 0 to 3 B5 - B0: song number, 1 to 61 SONGNO = 62, snd effect adr in next 2 bytes		
1	the LSB of an address...								usually, the addr of the next note in song; if SONGNO = OFEH, this is the LSB of the addr of the special sound.effect routine
2	the MSB of an address...								usually, the addr of the next note in song; if SONGNO = OFEH, this is the MSB of the addr of the special sound effect routine
3	F2	F3	F4	F5	F6	F7	F8	F9	bottom 8 bits of 10 bit freq data
4	ATN:CTRL or ATN:0 0 F0 F1								if CH# = 0 (noise): ATN ! 0 FB SHIFT if CH# = 1 - 3 (tone): MSN = 4 bit ATN, LSN = top 2 bits freq (0 0 F0 F1)
5	NLEN								determines duration of note: if freq swept, = # of steps in the sweep if not, NLEN * 16.7ms = duration of note
6	FPS		FPSV		freq sweep duration prescaler: FPS = prescaler reload value FPSV = temp FPS variable storage				
7	FSTEP								freq sweep step size: 1 to 127, -1 to -128 if FSTEP = 0, freq is not to be swept
8	ASTEP		ALEN		ALEN = # steps in atnswp: 2 - 15 (0 => 16) ASTEP = step size: 1 to 7, -1 to -8 if whole byte = 00, atn not to be swept				
9	APS		APSV		atn duration prescaler: APS = prescaler reload value APSV = temp APS variable storage				

DURATIONS:

fixed frequency = $NLEN \times 16.7ms$
frequency sweep = $[(NLEN - 1) \times FPS] + \text{initial FPSV} \times 16.7ms$
duration 1st step = $\text{initial FPSV} \times 16.7ms$
duration all others = FPS
FPS: 0 to 15 (0 => 16)
FPSV: 0 to 15 (0 => 16)
NLEN: 0 to 255 (0 => 256)
attenuation sweep = $[(ALEN - 1) \times APS] + \text{initial APSV} \times 16.7ms$
duration 1st step = $\text{initial APSV} \times 16.7ms$
duration all others = APS
APS: 0 to 15 (0 => 16)
APSV: 0 to 15 (0 => 16)
ALEN: 0 to 15 (0 => 16)

Note Header

FIGURE 2

Length in bytes: 1

Location: begins each block of 1 to 10 bytes of note data in CART ROM

Offset	B7	B6	B5	B4	B3	B2	B1	B0	Description
--------	----	----	----	----	----	----	----	----	-------------

*** REST

0	CH#	1	1	duration	B7 - B6	= channel number, 0 - 3	B4 - B0	= duration, 1 to 30	
---	-----	---	---	----------	---------	-------------------------	---------	---------------------	--

or, if B5 = 0, header precedes note data or is special indicator:

*** NOTE

0	CH#	0	0	0	0	0	type	B7 - B6	= channel number, 0 - 3
								B1 - B0	= note type, 0 - 3.

or

*** END OF SONG / REPEAT SONG

0	CH#	0	1	1	R	0	0	0	0
									if B4 = 1, end of song on channel in B7-B6:
									if B3 = 1, repeat song forever
									if B3 = 0, don't repeat

or

*** SPECIAL EFFECT

0	CH#	0	0	0	1	0	0	0	0
									this note is to be "played" by a special
									sound effect routine whose addr is
									contained in the following 2 bytes

REST DURATION = duration * 16.7ms
duration: 1 to 31

FIGURE 3

LST_OF_SND_ADDRS

Description: LST_OF_SND_ADDRS is a list of the starting addresses of each song's data area and note list. They are used by JUKE_BOX as source (note list) and destination (song data area) pointers. Each song's entries are stored as follows:

Byte 1: LSB of the address of the start of that song's note list
 Byte 2: MSB of the address of the start of that song's note list
 Byte 3: LSB of the address of the start of that song's data area
 Byte 4: MSB of the address of the start of that song's data area

The beginning address of LST_OF_SND_ADDRS is stored in a dedicated CPU RAM 16 bit word, PTR_TO_LST_OF_SND_ADDRS (xxxxH), allowing the cartridge programmer to place LST_OF_SND_ADDRS wherever desired.

NOTE: In other data structures, six bits are allocated for the song number (SONGNO). However, song numbers 0, 62, and 63 are used as special indicators, leaving song numbers 1 - 61 available. Therefore, the first entry in LST_OF_SND_ADDRS is for song number 1.

Length in bytes: 4 * total number of songs

Location: CART ROM

Beginning address: pointed to by CPU RAM word \$LST_OF_SND_ADDRS (xxxxH)

		Contents									
Offset		B7	B6	B5	B4	B3	B2	B1	B0	Description	
0		LSB								of starting adr of the note list for song number 1	
1		MSB								of starting adr of the note list for song number 1	
2		LSB								of starting adr of the data area for song number 1	
3		MSB								of starting adr of the data area for song number 1	
4		LSB								of starting adr of the note list for song number 2	
etc...											
4 * n		MSB								of starting adr of the data area for song number n (n = total number of songs)	

Rests

FIGURE 4

Length in bytes: 2

Location: CART ROM

Beginning address: pointed to by bytes 1&2 in that song's data area

	Contents								
Offset	B7	B6	B5	B4	B3	B2	B1	B0	Description
0	CH# 1 duration								if B7 = 1, header describes a rest: B4 - B0 = duration, 1 to 30

REST DURATION = duration * 16.7ms
 duration: 1 to 30

Note Type 0:
 fixed frequency, fixed attenuation

Length in bytes: 4

Location: CART ROM

Beginning address: pointed to by bytes 1&2 in that song's CPU RAM data area

	Contents								
Offset	B7	B6	B5	B4	B3	B2	B1	B0	Description
0	CH# 0 0 0 0 0 0 0								header
1	F2 F3 F4 F5 F6 F7 F8 F9								least significant 8 bits of 10 bit freq data
2	ATN				0	0	F0	F1	ATN = 4 bit atn data, LSN = top 2 bits freq
3	NLEN								NLEN * 16.7ms = duration of note

NOTE DURATION = NLEN * 16.7ms
 NLEN: 1 to 255 (0 => 256)

Note Type 1:
swept frequency, fixed attenuation

Length in bytes: 6

Location: CART RDM

Beginning address: pointed to by bytes 182 in that song's CPU RAM data area

Contents	
Offset : B7 B6 B5 B4 B3 B2 B1 B0	Description
0	CH# : 01 01 01 01 1 : header
1	F2 F3 F4 F5 F6 F7 F8 F9 : least sig 8 bits of initial 10 bit freq.data
2	ATN : 0 0 F0 F1 : ATN = 4 bit atn data, LSN. = top 2 bits freq
3	NLEN : NLEN = number of steps in the sweep
4	FPS FPSV : freq sweep duration prescaler: FPS = prescaler reload value FPSV = initial FPSV
5	FSTEP : freq sweep step size: 1 to 127, -1 to -128

NOTE DURATION = $[(NLEN - 1) * FPS] + \text{initial FPSV} * 16.7\text{ms}$

duration 1st step = $\text{initial FPSV} * 16.7\text{ms}$

duration all others = FPS

FPS: 0 to 15 (0 => 16)

FPSV: 0 to 15 (0 => 16)

NLEN: 0 to 255 (0 => 256)

Note Type 2:

FIGURE 6

fixed frequency, swept attenuation

Length in bytes: 6

Location: CART ROM

Beginning address: pointed to by bytes 182 in that song's CPU RAM data area

		Contents								
Offset	B7	B6	B5	B4	B3	B2	B1	B0	Description	
0	CH# 01 01 01 01 11 01								header	
1	F2 F3 F4 F5 F6 F7 F8 F9								least significant 8 bits of 10 bit freq data	
2	ATN		00		F0 F1		ATN = init atn data, LSN = top 2 bits freq			
3	NLEN				NLEN * 16.7ms = duration of note					
4	ASTEP		ALEN		ALEN = # steps in atnswp: 2 - 15 (0 => 16) ASTEP = step size: 1 to 7, -1 to -8 if whole byte = 00, atn not to be swept					
5	APS		APSV		atn duration prescaler: APS = prescaler reload value APSV = initial APSV					

NOTE DURATION = NLEN * 16.7ms
NLEN: 1 to 255 (0 => 256)

ATN SWEEP DURATION = [((ALEN - 1) * APS) + initial APSV] * 16.7ms
duration 1st step = initial APSV * 16.7ms
duration all others = APS
APS: 0 to 15 (0 => 16)
APSV: 0 to 15 (0 => 16)
ALEN: 0 to 15 (0 => 16)

Note Type 3:

swept frequency, swept attenuation

Length in bytes: 8

Location: CART ROM

Beginning address: pointed to by bytes 182 in that song's CPU RAM data area

Contents		Description
Offset	B7 B6 B5 B4 B3 B2 B1 B0	
0	CH# 0 0 0 0 1 1 1	header
1	F2 F3 F4 F5 F6 F7 F8 F9	least sig 8 bits of init 10 bit freq data
2	ATN 0 0 F0 F1	ATN = init atn data, LSN = top 2 bits freq
3	NLEN	NLEN = number of steps in the sweep
4	FPS FPSV	freq sweep duration prescaler: FPS = prescaler reload value FPSV = initial FPSV
5	FSTEP	freq sweep step size: 1 to 127, -1 to -128
6	ASTEP ALEN	ALEN = # steps in atnswp: 2 - 15 (0 => 16) ASTEP = step size: 1 to 7, -1 to -8 if whole byte = 00, atn not to be swept
7	APS APSV	atn duration prescaler: APS = prescaler reload value APSV = initial APSV

NOTE DURATION = $[(NLEN - 1) * FPS] + \text{initial FPSV}] * 16.7\text{ns}$
 duration 1st step = $\text{initial FPSV} * 16.7\text{ns}$
 duration all others = FPS
 FPS: 0 to 15 (0 => 16)
 FPSV: 0 to 15 (0 => 16)
 NLEN: 0 to 255 (0 => 256)

ATN SWEEP DURATION = $[(ALEN - 1) * APS] + \text{initial APSV}] * 16.7\text{ns}$
 duration 1st step = $\text{initial APSV} * 16.7\text{ns}$
 duration all others = APS
 APS: 0 to 15 (0 => 16)
 APSV: 0 to 15 (0 => 16)
 ALEN: 0 to 15 (0 => 16)

**Noise notes:
Special case type 2 notes**

FIGURE 8

Length in bytes: 5
Location: CART ROM
Beginning address: pointed to by bytes 1&2 in that song's CPU RAM data area

		Contents																	
Offset	:	B7	B6	B5	B4	B3	B2	B1	B0	Description									

0	:	1	0	:	0	:	0	:	0	:	0	:	1	:	1	:	0	:	header (CH# = 0, indicates noise note)

1	:	ATN				:	0	:	FB SHIFT				:	MSN = 4 bit noise ATN data (init if swept) LSN = noise control data (SHIFT = NF0 NF1)					

2	:	NLEN								:	NLEN * 16.7ns = duration of note								

3	:	ASTEP				:	ALEN				:	ALEN = # steps in atnswp: 2 - 15 (0 => 16) ASTEP = step size: 1 to 7, -1 to -8 if whole byte = 00, atn not to be swept							

4	:	APS				:	APSV				:	atn duration prescaler: APS = prescaler reload value APSV = temp APS variable storage							

NOTE DURATION = NLEN * 16.7ns
NLEN: 1 to 255 (0 => 256)

ATN SWEEP DURATION = [((ALEN - 1) * APS) + initial APSV] * 16.7ns
duration 1st step = initial APSV * 16.7ns
duration all others = APS
APS: 0 to 15 (0 => 16)
APSV: 0 to 15 (0 => 16)
ALEN: 0 to 15 (0 => 16)

FIGURE 9

Dedicated cartridge RAM locations and Special Effect format

Length in bytes: 11
 Location: CPU RAM
 Beginning address: 7020H

PTR_TO_LST_OF_SND_ADDRS DS 2 ;pointer to start of LST_OF_SND_ADDRS

PTR_TO_S_ON_1 DS 2 ;pointer to data area of song to be played on CH# 1
 PTR_TO_S_ON_2 DS 2 ;pointer to data area of song to be played on CH# 2
 PTR_TO_S_ON_3 DS 2 ;pointer to data area of song to be played on CH# 3
 PTR_TO_S_ON_0 DS 2 ;pointer to data area of song to be played on CH# 0

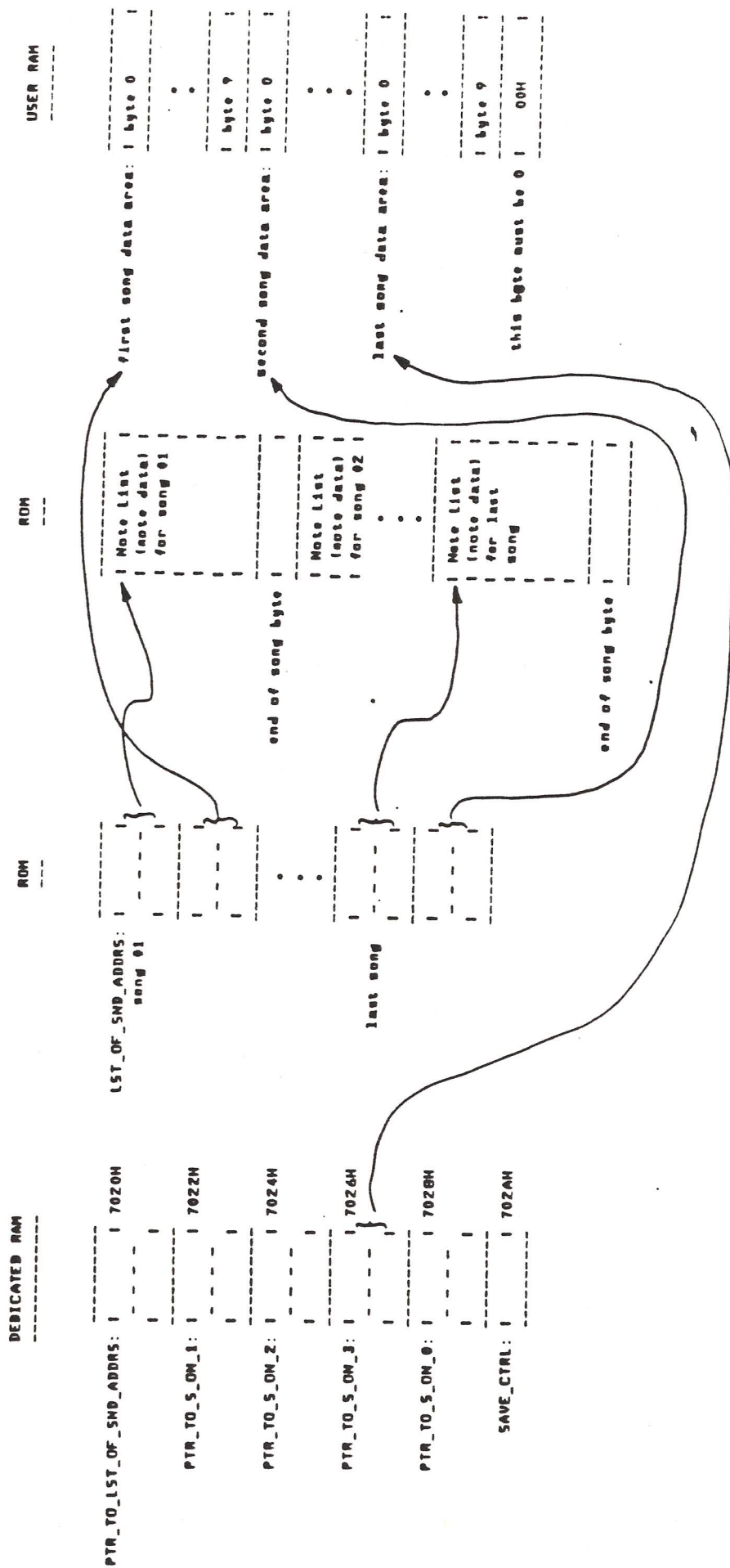
SAVE_CTRL DS 1 ;LSN = last control data sent to noise generator

Special Effect format

All special effect routines should be written in the following format
 (SFX = the address of the effect routine, stored in ROM after the effect's header, IX is passed pointing to the song's data area):

```
SFX:  LD (SAVE_x_NNP),HL      ;save address of next note in song
      LD (SAVE_x_SONGNO),A   ;save song's SONGNO
      RET                   ;to LOAD_NEXT_NOTE
SFX+7: LD HL,SAVE_x_SONGNO   ;test for 1st pass through effect
      BIT 7,(HL)
      JR NZ,NOT_PASS_1
      SET 7,(HL)            ;to prevent further passes thru init's
      ...                  ;initialize bytes within the data area here
      RET                   ;to LOAD_NEXT_NOTE
NOT_PASS_1: ...              ;code for pass 2 or greater starts here,
      .                    ;which algorithmically modifies freq, atn,
      .                    ;or control data within song data area
      .                    ;pointed to by IX
      RET (to PROCESS_DATA_AREA) if effect not over
      ;if here, effect is over, so restore SONGNO and addr next note in song
      LD HL,(SAVE_x_NNP)    ;HL := addr next note in song
      LD DE,SAVE_x_SONGNO   ;DE := addr saved song number
      CALL LEAVE_EFFECT     ;to restore them to bytes 0 - 2 in data area
      JP EFXOVER            ;in PROCESS_DATA_AREA to load song's next note
```


FIGURE 10



NOTES AND ERRATA
for ColecoVision Sound Users' Manual
Version 1.1

The following is a list of known errors in the User's Manual and a set of explanatory notes which hopefully serve to clarify some of the concepts presented therein. It is suggested that, prior to reading the Manual, corrections be made according to the following list and references to the explanatory notes be marked at the appropriate passages.

*** ERRATA ***

- page 2 about half way down the page, starting near the right margin:
 " where x = the song's SONGNO"
 - should read -
 delete - There is no necessary relationship between a SONGNO and a song data area. As becomes clear later, several songs (with different SONGNOs) can share the same data area. This error also occurs in the first sentence of FIGURE 1.
- page 12 next to last line:
 - add -
 "0 = no sweep"
- page 16 first line:
 "PLAYSONGS"
 - should read -
 "PLAY_SONGS"
- page 18 last two lines:
 "RETS C SET..."
 "RETS C RESET..."
 - should read -
 "RETS Z SET..."
 "RETS Z RESET..."
- page 21 the following utilities are not generally useful, are not available as entry points, and therefore can not be used:
 AREA_SONG_18
 UPATNCTRL
 UPFREQ
 PT_IX_TO_SxDATA
 UP_CH_DATA_PTRS (on page 22)
- FIG 2 last line of REST description:
 "B4 - B0 = duration, 1 to 31"
 - should read -
 "B4 - B0 = duration, 1 to 30"
- FIG 3 about one third down and half way down the page:
 "xxxxH"
 - should read -
 "7020H"

USER RAM

This is the area in CRAM that the cartridge programmer has chosen to hold the ten byte song data areas which contain sound and timing information to be processed by the OS sound routines. These data areas must be stored as contiguous blocks of ten bytes each. In all cases but one, the programmer may choose to "play" a song in any data area; however, song #1 MUST use the FIRST song data area for the OS routines to work properly. Also, the byte immediately following the last byte in the last data area MUST be zero (this code tells the OS routine SND_MANAGER to "stop looking for more song data areas; see bottom of page 5, Users' Manual). This byte will automatically be set to zero by proper invocation of the INIT_SOUND routine before any other OS sound routines are called (see pages 5 and 13).

2) The ColecoVision OS entry point names of some of the sound routines and dedicated locations are different from their names given in the Sound Users' Manual. They are:

Entry Point -----	Sound Users' Manual -----
PLAY_IT	JUKE_BOX
SOUND_MAN	SND_MANAGER
SOUND_INIT	INIT_SOUND
NOTES	PTR_TO_LST_OF_SND_ADDRS

You should use the entry point names.

3) Page 22, last paragraph: It is mentioned that a special effect routine may want to call the OS sound routines FREQ_SWEEP and ATN_SWEEP to operate upon data within the effect's data area, which "require that data be ordered appropriately within a song data area". This means:

Whether or not the special effect uses FREQ_SWEEP or ATN_SWEEP: bytes 3 and 4 (see FIGURE 1) MUST contain the frequency and attenuation data as specified. This is because PLAY_SONGS (called every interrupt) will output

For FREQ_SWEEP used by itself - in addition to bytes 3 and 4, bytes 5, 6, and 7 must contain data as specified. Bytes 8 and 9 may be used for whatever (since FREQ_SWEEP doesn't look at them).

For ATN_SWEEP used by itself - in addition to bytes 3 and 4, bytes 5, 8, and 9 must contain data as specified. Bytes 6 and 7 may be used for whatever (since ATN_SWEEP doesn't look at them).

If both FREQ_SWEEP and ATN_SWEEP are used, all bytes in the data area must look as specified in FIGURE 1.

PTR_TO_S_ON_3:

7026-27H - As above, for tone generator #3.

PTR_TO_S_ON_0:

7028-29H - As above, for tone generator #0 (the noise generator).

The final byte at 702AH, SAVE_CTRL, is used by the OS sound routines to store data necessary for smooth operation of the noise generator (see bottom of page 11 in the Users' Manual).

All 11 bytes should be initialized before the OS routines which operate upon them are called: this is done by calling INIT_SOUND and passing the appropriate cartridge-dependent information (see Users' Manual pages 5 and 13).

ROM

Cartridge ROM to be used by OS sound routines is divided into two sections: LST_OF_SND_ADDRS and the Note List.

LST_OF_SND_ADDRS:

A contiguous list of 4 bytes per each song (or special effect) used by the game. In each 4 byte section, the first 2 bytes are a pointer to the beginning of the song's note list (also in ROM). The second two bytes are a pointer to the song data area in RAM to be used by that song (review pages 2 and 3 in the Manual). Of course, another song may also use the same data area, since there can be (and usually are) more songs than there are data areas. NOTE, however, that Song #1 MUST be the first entry in LST_OF_SND_ADDRS for the OS routines to operate properly.

-- EXAMPLE -- The first two bytes in this list are a pointer to song #1's note list, as they MUST be. The second two bytes are a pointer to the song's data area in RAM, shown here pointing to the first song data area as also MUST be the case (see later). As can also be seen from the figure, the last song happens to use the second song data area.

In summary, there is a note list for every note list pointer in LST_OF_SND_ADDRS, but, since songs may share RAM data areas, there are almost always more data areas than there are data area pointers.

NOTE LIST:

The Note List is a contiguous block of ROM containing the data which comprise the notes of each song. The number of bytes per song of course varies with the length of the song. The first byte of the note list for a song is pointed to by a two byte entry in LST_OF_SND_ADDRS (see above). The last byte of each song's note list is a single byte end of song/repeat code (see page 2 and Figure 2 in the Users' Manual).

also, half way down the page:
"...pointed to by CPU RAM word LST_OF_SND_ADDRS"
- should read -
"...pointed to by CPU RAM word PTR_TO_LST_OF_SND_ADDRS"

FIG 4 second line:
"Length in bytes: 2"
- should read -
"Length in bytes: 1"

also, about 6 lines down from that:
"B4 - B0= duration, 1 to 31"
- should read -
"B4 - B0= duration, 1 to 30"

*** NOTES ***

1) After reading through page 6 in the Users' Manual, it may be helpful to review the following summary of dedicated pointers and data structures (discussion refers to Figure 10, included as part of these notes):

DEDICATED RAM

Prior to calling any OS sound routines (except INIT_SOUND), the 11 CRAM locations 7020H through 702AH must be initialized to meaningful data. Ten of the locations are two byte pointers:

PTR_TO_LST_OF_SND_ADDRS:
7020-21H - Points to the start of a list of pointers in cartridge ROM, LST_OF_SND_ADDRS (see later). The OS sound routines know where the cartridge-dependent LST_OF_SND_ADDRS is stored through this pointer. It is shown pointing to the first byte in this ROM list (as it must).

PTR_TO_S_ON_1:
7022-23H - This and the following three pointers are used by OS sound routines to store the addresses of the four song data areas which currently contain the sound data to be modified/output to the four sound generators in the TI sound chip. This pointer stores the address for the song currently playing on tone generator #1.

-- EXAMPLE -- PTR_TO_S_ON_3 is shown pointing to the second song data area. I.e., data for the song currently playing on tone generator #3 happens to be stored in the second song data area. The second data area is used for purposes of illustration only: other songs may very well require that data for tone generator #3 be stored in a different song data area.

PTR_TO_S_ON_2:
7024-25H - As above, for tone generator #2.

APPENDIX D

DIRECTORY OF COLECOVISION SOFTWARE BULLETINS

0001..... Colecovision Software Bulletin
0002..... Error in Write_Vram Routine
0003..... OS Bug - PTR_TO_LST_OF_SND_ADDR in wrong place
0004..... Technique - Turning off songs without going into the tables.
0005..... Bug in OS Activate routine
0006..... Release of Additional OS Entry Points
0007..... Header:UTL
0008..... Music Tables
0009..... Songbird File
0010..... Interrupt Handling Routines

TO: DISTRIBUTION

cc: Eric Bromely
Marshall Caras
Robert Schenck

FROM: DAVID HWANG

SUBJECT: COLECOVISION SOFTWARE BULLETIN

The Colecovision Software Bulletin has been set up to assist Colecovision programming users to understand, maintain and develop the system/application software.

More specifically, its purposes are:

- (1) Part of the continuing effort to document the operating system (currently OS_7:OS);
- (2) Keep users updated regarding any patches and revisions of the operating system;
- (3) Function as a user's library for information exchange. Any proven routines or modules which can be used as tools to facilitate software development will be properly documented here with author(s) duly credited.

BULLETIN NO. 0002

May 25, 1982

TO: DISTRIBUTION
FROM: Z. SMITH/D. HWANG
SUBJECT: ERROR IN WRITE_VRAM ROUTINE

cc: Eric Bromley
Marshall Caras
Robert Schenck

WRITE_VRAM has a problem:

- It works as advertised for byte counts less than 100H and for byte counts that are even multiples of 100H. For other values, it subtracts 100H from the actual byte count that is written.
- Cartridge programmers should deal with this problem (and corresponding problems it will cause in any OS routine that writes VRAM, except for WR_SPR_NM_TBL) by always sending numbers of bytes that are less than or even multiples of 100H.
- They should not deal with it by padding their byte counts as this may lead to cartridges that fail when the bug is fixed.

BULLETIN NO. 0003

June 7, 1982

TO: DISTRIBUTION

cc: Eric Bromley
Marshall Caras
Robert Schenck

FROM: Z. SMITH/D. HWANG

SUBJECT: ERROR IN OS SOUND PACKAGE

There is a bug in the OS sound software:

- The data structure PTR_TO_LST_DF_SND_ADDR, which takes up 11 RAM bytes, is not located in OS RAM above 73BAH as it should be, but instead has been placed in the cartridge programmer's RAM at 7020H. Cartridge programmers should avoid using RAM from 7020H thru 702AH when the sound software is in operation.

BULLETIN NO. 0004

June 7, 1982

TO: DISTRIBUTION

cc: Eric Bromley
Marshall Caras
Robert Schenck

FROM: Z. SMITH/D. HWANG

SUBJECT: A TECHNIQUE FOR TURNING OFF SONGS AND SOUNDS

The need has arisen for a safe way of turning off an individual "song" or sound event before its time. The most obvious method, which involves writing OFFH to the first byte of that song's sound area is not recommended, since it could lead to incompatibility later on if, for any reason, we wanted to change the length of the sound area data structure.

The safest method that I have been able to come up with is to declare a "null song" for each sound area which consists of nothing more than a non-repeating 64'th rest. If this song is allowed to be the highest priority song for a given area, playing it will have the effect of turning off whatever sound happened to be playing in that area previously.

6/18/82

TO: DISTRIBUTION

cc: Eric Bromley
Marshall Caras
Robert Schenck

FROM: Z. Smith/D. HWANG

SUBJECT: BUG IN OS ACTIVATE ROUTINE

There is a bug in the OS Activate routine which surfaces when Activate is called on a Semi-Mobile object in Graphics Mode 1.

In this mode, Activate writes the pattern generators for a Semi-Mobile object to VRAM properly, but miscalculates the number and placement in VRAM of the corresponding color bytes when operating on generators in the upper half of the stable.

This leads to 2 problems:

- The upper half of the color table is not written by Activate.
- The color bytes intended for this half of the table are written elsewhere in VRAM possibly overwriting some other table.

Cartridge programmers should avoid using Activate to write pattern generators to VRAM in Graphics Mode 1 whenever possible. Or, if it is absolutely necessary to use Activate in this way they should count, first of all, on having to write the color table separately, and second, on guarding against the second problem listed above by isolating the color table.

SEPTEMBER 17, 1982

TO: DISTRIBUTION

FROM: K. LAGACE/D. HWANG

SUBJECT: RELEASE OF ADDITIONAL OS ENTRY POINTS
OS_SYMBOLS:OS REV. 1

cc: Eric Bromley
Marshall Caras
Robert Schenck

Module Name	Address	Description	Inputs	Outputs	Regs. Destroyed
ADD816	001B1H	Adds 8 bit signed number in "A" to 16 bit number pointed to by "HL".	- 8 bit # in A - 16 bit # addr in HL	Altered 16 bit # at HL addr.	A,F,B
DECLSN	00190H	Decrements LSN of byte pointed to by "HL" without affecting MSN or "HL".	- 8 bit # addr. in HL	Altered 8 bit # at HL addr. Z flag if 0 C flag if -1	A,F
DECMN	00198H	Decrements MSN of byte pointed to by "HL" without affecting LSN or "HL".	- 8 bit # addr. in HL	Altered 8 bit # at HL addr. Z flag if 0 C flag if -1	A,F
OLSN	001A6H	Copies MSN of byte pointed to by "HL" to LSN of that byte.	- 8 bit # addr. in HL	MSN/LSN of # at HL addr. becomes MSN/MSN	A,F,B

FOR SOUND USE ONLY

ATN_SWEEP	0012FH	Creates attenuation sweeps by altering attenuation data stored in song data area.	See Sound Users Manual	See Sound Users Manual	All
FREQ_SWEEP	000FCH	Creates frequency sweeps by altering frequency data stored in song data area.	See Sound Users Manual	See Sound Users Manual	All
EFXOVER	002EEH	See Sound Users	See Sound Users	See Sound Users	All
LEAVE_EFFECT	001D5H	See Sound Users	See Sound Users	See Sound Users	All

BULLETIN NO. 0007

OCTOBER 21, 1982

TO: DISTRIBUTION
FROM: ARD SOFTWARE ENGINEERING
SUBJECT: HEADER:UTL REV. 3

cc: Robert Schenck

Released and supported on HP64000 systems under USERID:UTL is the new revision of HEADER which has been adopted by Software Engineering as part of standard documentation for any module and program developed in house (Ref. R. Jepson's memo July 27, 1982).

The SETHEAD:UTL has also been updated to support HEADER Rev. 3

MEMORANDUM

NO. 0008
OCTOBER 27, 1982

TO: DISTRIBUTION
FROM: MUSIC AND SOUND DEPT./D. HWANG
SUBJECT: MUSIC TABLES

cc: Robert Schenck

LST_OF_SND_ADDRS has formerly been used in all games to denote the starting address of a list of pointers to song tables and work areas. This label will not be used in the future games. Instead a label with postfix "NOTES" will be used.

For example, in the upcoming games:

DONKEY KONG JR	will use	KONGJRNOTES
OMEGA RACE	will use	OMEGANOTES
GORF	will use	GORFNOTES

MEMORANDUM

NO. 0009
OCTOBER 27, 1982

TO: DISTRIBUTION
FROM: MUSIC AND SOUND DEPT./D. HWANG
SUBJECT: SONGBIRD FILE

cc: Robert Schenck

Effective immediately all work pertaining to music and sounds will be done in the SONGBIRD file. To play a song, a call to a descriptive label, which is supplied by the music group, will be used. For example:

CALL BELL_SOUND

Where BELL_SOUND is a global label in the SONGBIRD file which will contain all that is necessary to play that sound or song. This one call approach is replacing the former procedure such as:

```
LD      B,3      ; THE SONG NUMBER
CALL    PLAY_IT
LD      B,4
CALL    PLAY_IT
```

Within the SONG_BIRD file, song numbers will be EQUAT to descriptive labels instead of using absolute numbers.

BULLETIN NO. 0010

OCTOBER 27, 1982

TO: DISTRIBUTION
FROM: R. JEPSON/D. HWANG
SUBJECT: INTERRUPT HANDLING ROUTINES

cc: Eric Bromley
Marshall Caras
Robert Schenck

Due to the hardware configuration of ColecoVision, certain routines which write to VRAM, such as WRITE-VRAM, could have undetermined results if they are interrupted during execution.

One possible solution to this problem is a defer interrupt routine. The function of this routine is to hold off (defer) the servicing of the interrupt until the cartridge program believes it is safe to service it.

An example of a defer interrupt service is attached using a binary semaphore type construct. Note that the routines do not stop the actual interrupt (they can't) but just hold off processing the interrupt until later.

The key point to the routine is that another interrupt cannot occur until the VDP status register is read.

An example of the use of DEF_INT:

```
LD      IX,WRITE_VRAM      ;IX gets routine that the
                           ;cartridge program does
CALL    DEF_INT            ;not want interrupted
```

SOURCE LINE

1 "08Z"

3

25

Q

• 2

5

9

1

2

1

3

5

2

0

2

5

9

□

9

• 2

24

3

LOCATION OBJECT CODE LINE SOURCE LINE

```
59 10Outputs;
60 GLB NMI_INTERRUPT
61 GLB DEF_INT
62
63
64 jCalls;
65 routine passed in IX in the case of DEF_INT
66 jOS Calls
67 EXT VDP_STATUS_BYTE
68 EXT READ_REGISTER
69
70 COMMENT MACRO
71 .GOTO ENDCOMMENT
72
73
74
75 DEF INT
76
77
78
79 Pseudo code
80
81 Procedure DEF_INT ('OS_routine');
82 Push AF (In case it has an input parameter)
83 INT_DEF_BYTE := DEFER; (set bit 0)
84 Pop AF (With input parameter to OS routine)
85 Save HL (In case of input parameter)
86 HL := Return address
87 Exchange Return address with saved HL value
88 Gate OS routine passed as input (in IX)
89
90
91
92
93
94
95
96
97
98
99
100 END; (DEF_INT)
101
102 ENDCOMMENT;
103 MEND
104
105 *****
106
107 PROC
```

LOCATION OBJECT CODE LINE SOURCE LINE

```

130 COMMENT2 MACRO
131 .GOTO ENDCOMMENT
132
133

```

```

134 **** NMI_INTERRUPT **** ROL
135 ****
136 ****
137 ****
138 ****
139 ****

```

We get here whenever VDP interrupts (every 1/60 sec.). The deferred VRAM writes take place here and the interrupt driven sound, keyboard scan and timer routines are also exercised. At the end of the routine, the VDP status register is read and its contents are saved in STAT_SAVE.

Before the routine runs, it checks interrupt defer bit of INT_DEF_BYTE (bit 0). If this bit is set, it is an indication that a routine is in progress which accesses the VDP and therefore should not be interrupted by another routine which also accesses the VDP. Interrupt occurred bit of INT_DEF_BYTE (bit 7) is then set and the routine returns without performing any other functions.

Pseudo code

```

151
152
153 Procedure NMI_INTERRUPT
154   Save Accumulator
155   Save all registers except Accumulator
156   Save alternate registers
157   If DEFER in INT_DEF_BYTE (bit 0)
158     then INT_DEF_BYTE := INT_DEF_BYTE + [INTERRUPT_OCCURRED]; Set bit 7
159   else
160     Begin
161       Process interrupt routines
162     End;
163   Restore alternate registers
164   Restore all registers except Acc
165   Read VDP status register
166   Restore Acc
167
168 End; (NMI_INTERRUPT)
169
170
171 END_COMMENT;
172 MEND
173
174 PROG

```


BULLETIN NO. 11
DECEMBER 22, 1982

TO: DISTRIBUTION

FROM: ARD SOFTWARE ENGINEERING *David J. Haring*
SUBJECT: RELEASE OF COLECOVISION PROGRAMMERS
MANUAL REV. 5

cc: Eric Bromley
Robert Schenck
Marshall Caras
Tom Helmer

The ColecoVision Programmer's Manual Rev. 5 has been released. This manual is written for the applications programmer and is intended as both a day-to-day reference source as well as a training document for programmers new to ColecoVision.

This new edition contains the overview for both hardware and software. Subsequently, detail descriptions are given in the areas of:

- Graphics Generation Software
- Interrupt Handling
- Timing
- Controller Software
- Sound Generation Software
- Boot up Software and Utilities
- Defined Reference Locations

The Rev. 5 manual pertains to the current production OS_7. Fundamental knowledge of the OS is presented in the manual without elaborating on application examples and design approaches. These materials will be documented in the proposed ColecoVision Applications Manual, scheduled to be released in second quarter 1983.

In the Appendix B you will find the graphics documentation (Rev. 1.0) has been updated with addition of materials describing PUT_SPRIT and PUT_COMPLEX.

The Sound documentation also received updates in the form of Notes and Errata attached at the end of Appendix C.

User feedback should be addressed to the Manager of Software Engineering of Coleco ARD. All adopted changes will be brought to your attention via ColecoVision Bulletin announcements.

This manual is confidential and should not be copied. All releases have to be signed out through the ARD Engineering secretary S. Rakowski.

DISTRIBUTION:

C. Baldyga	K. Lagace
R. Dionne	J. Michaels
A. Godfrey	M. Minto
L. Gray	A. Nguyen
C. Hager	L. Olbrych
R. Harris	D. Schulze
R. Jepson	D. Stern
D. Jonker	D. Thompson

CONFIDENTIAL



Executive Offices:

INDUSTRIES, INC. 845 ASYLUM AVENUE HARTFORD, CT 06105 (203) 278-0280

ColecoVision Software Bulletin

BULLETIN NO. 0012

March 17, 1983

TO: DISTRIBUTION

FROM: ARD SOFTWARE ENGINEERING

DKH KAL

RE: CORRECTIONS IN REGARD TO BULLETIN NO. 0004

-
- (1) The statement that "Sound Data Areas are off limits to programmers" is not true.
 - (2) The "Null Song" method wastes CROM space. Writing OFFH to the first byte of the song's sound area IS recommended.

Since the ColecoVision Operating System turns off sounds by placing OFFH into the first byte of the Sound Data Areas anyway and changing the data structures of the Sound Data Areas would entail changing the operating system. It has been proven that the above method is the fastest and most direct way to abort sounds.

The "null song" method may still be used, but each additional song uses at least five bytes of CROM; four for the LST_OF_SND_ADDRS and one for the END code.

DISTRIBUTION:

C. Baldyga	K. Lagace
R. Dionne	J. Michaels
A. Godfrey	M. Minto
L. Gray	A. Nguyen
C. Hager	L. Olbrych
R. Harris	R. Rizzo
L. Henderson	D. Schulze
R. Jepson	D. Stern
D. Jonker	D. Thompson
	B. Zawislak

CC: E. Bromley
R. Schenck
C. M. Caras
T. Helmer

CONFIDENTIAL

ColecoVision Software Bulletin

Bulletin No. 0013
April 4, 1983

TO: Distribution
FROM: ARD Software Engineering *DKH RFS*
RE: Release of Additional ColecoVision OS Entry Points

The following is a list of additional entry points to the ColecoVision OS ROM.

PX_TO_PTRN_POS	EQU	07E8H
PUT_FRAME	EQU	080BH
GET_BKGRND	EQU	0898H
CALC_OFFSET	EQU	08C0H

Attached is a brief description of the routines which correspond to the entry points released.

DISTRIBUTION

C. Baldyga	J. Michaels
R. Dionne	M. Minto
L. Gray	A. Nguyen
C. Hager	L. Olbrych
R. Harris	R. Rizzo
L. Henderson	B. Rochette
R. Jepson	D. Schulze
D. Jonker	D. Stern
K. Lagace	D. Thompson
	B. Zawislak

CC: E. Bromley
R. Schenck
D. Hwang
C. M. Caras
T. Helmer
File

CONFIDENTIAL

Here are the graphic subroutines which would be useful to have access to, along with a brief description of what each one does.

PX_TO_PTRN_POS (Pixel to pattern plane position)
(entry point xxxxH)

This routine divides the 16 bit signed value in the DE register pair by 8. An 8 bit signed result is returned in register E. Results of less than -127 are returned as -128, results of greater than +126 are returned as +127.

If this routine is passed the X(or Y) pixel coordinate position of a point on the pattern plane, the X(or Y) coordinate in pattern positions will be returned.

INPUT: DE = N (16 bit signed number)

OUTPUT: N/8 < -128 E = -128
-128 <= N/8 <= 127 E = N/8
N/8 > +126 E = +127

REGISTERS AFFECTED:

FLAGS
DE

PUTFRAME
(entry point xxxxH)

CONFIDENTIAL

PUTFRAME moves data from cpu RAM to the Pattern Name Table in URAM. The data is assumed to be an array of Pattern Generator Names which when moved to the Pattern Name Table, will produce a rectangular graphic, or frame, composed of the patterns specified by these Pattern Generator Names. The array must be arranged in row major order.

The dimensions of array are passed to the routine in the BC register pair. These dimensions also define the height and width (in pattern plane positions) of the frame when displayed.

The upper left corner of the frame will appear on the pattern plane at a position determined by Y_PAT_POS and X_PAT_POS which are passed in the DE register pair. Y and X_PAT_POS are row and column coordinates in pattern plane positions as measured from the upper left corner of the pattern plane. Y and X_PAT_POS are interpreted as 8 bit signed values and, therefore, the corner of the frame may be placed anywhere within or outside the boundaries

of the pattern plane. Therefore, the frame itself may be placed partially off screen in any direction.

The HL register pair must contain the address of the start of the array of pattern names.

INPUT: HL = Address of array in CPU RAM
 B = Y dimension of array and Y_EXTENT of frame
 C = X dimension of array and X_EXTENT of frame
 D = Y_PAT_POS of upper left corner of frame
 E = X_PAT_POS of upper left corner of frame

OUTPUT: Modifies URAM name table

REGISTERS AFFECTED:

All registers used

As an example, if an array exists in CPU memory space which looks like...

ARRAY: DB 0,1,2,3,4,5

and the first six pattern generators in URAM have been initialized with the following patterns...

Pattern Generator #	Graphic
0	A
1	B
2	C
3	D
4	E
5	F

Then the following code sequence...

```
LD HL,ARRAY
LD B,2      ;B := Y_EXTENT
LD C,3      ;C := X_EXTENT
LD D,2      ;D := Y_PAT_POS
LD E,-1     ;E := X_PAT_POS
CALL PUT_FRAME
```

will produce this display...

```

      0 X_PAT_POS ->
Y_PAT_POS  . . . . .
V 0.      . . . . .
      . . . . .
      .B.C. . . . .
      .E.F. . . . .
      . . . . .
```

(diagram of upper left corner of pattern plane)

CONFIDENTIAL

Note: Patterns A and D are not seen, since they would be to the left of the left-hand edge of the pattern plane.

GET_BKGRND
(entry point xxxxH)

This routine is the inverse of the PUT_FRAME routine described above. GET_BKGRND moves an array of names from the pattern name table in URAM into CPU RAM. The dimensions of the array and the position of the upper left corner of the frame it defines, are passed to the routine in same manner as in PUT_FRAME. The names are moved to the location in CPU RAM specified by the contents of the HL register pair.

If part of the frame extends beyond the pattern plane, the names that correspond to positions which are not on the pattern plane will not be defined.

INPUTS: HL = Destination address in CPU RAM to which
 names will be moved
 B = Y_EXTENT of frame
 C = X_EXTENT of frame
 D = Y_PAT_POS of upper left corner of frame
 E = X_PAT_POS of upper left corner of frame

OUTPUTS: CPU RAM from HL to HL+(B*C)-1 filled with names
 from pattern name table

REGISTERS AFFECTED:

All registers used

CALC_OFFSET
(entry point xxxxH)

This routine calculates the offset from the start of the pattern name table corresponding to a pattern plane position specified by the coordinates Y_PAT_POS and X_PAT_POS.

The coordinates are passed to, and the result is passed back in the DE register pair.

INPUTS: D = Y_PAT_POS
 E = X_PAT_POS

OUTPUTS: DE = Offset from start of pattern name table

REGISTERS AFFECTED:

FLAGS
DE

CONFIDENTIAL

ColecoVison Software Bulletin

Bulletin No. 0014

April 12, 1983

TO: Distribution

FROM: ARD Software Engineering

RE: OS_SYMBOLS Rev.4

DKH

RFS

Attached please find a listing of OS_SYMBOLS Rev. 4. This listing holds all ColecoVision OS reserved data entry points released to date.

Attachment

Distribution:

C. Baldyga
R. Dionne
L. Gray
C. Hager
R. Harris
L. Henderson
R. Jepson
D. Jonker
K. Lagace
J. Michaels

J. Milano
M. Minto
A. Nguyen
L. Olbrych
R. Rizzo
B. Rochette
D. Schulze
D. Stern
D. Taylor
D. Thompson
B. Zawislak

CC:

E. Bromley
C. M. Caras
T. A. Helmer
D. Hwang
R. Schenck
File

LOCATION OBJECT CODE LINE SOURCE LINE

```

1 "Z80"
2
3 NAME 'Rev 4 - RFJ'
4
5 DESCRIPTION MACRO
6
7 .GOTO ENDESCRIPTION

```

```

8 Author: Zac Smith
9 User: OS
10 Starting date: 13may1982
11 Header Rev: 1

```

OS_SYMBOLS ColecoVision Operating System Software Engineering Advanced Research and Development Coleco Industries

*** Confidential ***

List of access points to the ColecoVision Operating system ROM. Only these points listed in this file have been approved as absolute locations of which the cartridge developer can access the OS ROM. Additionally, access to any memory locations indirectly, or by offset to locations defined herein is denied except where defined by the ColecoVision Programmer's Manual (current rev 15).

List of OS symbols in alphabetical order with defining and referencing modules (if any).

Rev History (one line note indicating the change)

Rev.	Date	Name	Change
4	13apr1359	Rob	Remove Zaxxon related documentation in preparation for re-release of this file for general distribution
	11apr1626	Rob	Added PUTFRAME (no underline) to match label in OS listing. Kept PUT_FRAME due to Software Bulletin released.
	11apr 900	Rob	Updated Header to expand the description of this file.
3	05apr1444	Rob	GLRed locations added in rev 3 Added location: PX_TO_PTRN_PDS PUT_FRAME
2	13sept1114p	Rob	GET_HKGRND COLC_OFFSET Added documentation specific to Zaxxon Development.
1	2sept11153p	Ken Logue	Added 9 SOUND OS equates
0	13may	Zac Smith	Initial Jump table equates
			DATE 1 5/13/82
			FOR REV : 5 (OS 5:05)

ENDESCRIPTION:
HEND

CONFIDENTIAL

LOCATION OBJECT CODE LINE SOURCE LINE

```

59
60 ;Symbol      Absolute      Partial Xref of routines used
61 ;Name        Address      by other OS routines
62 ;-----
63 ;Start of defined reference points
64 ACTIVATE      EQU 01FF7H
65 ACTIVATEP     EQU 01F64H
66 ADDR16       EQU 001R1H
67 AMERICA       EQU 00069H
68 ASCII_TABLE  EQU 00060H
69 ATN_SWEEP     EQU 0012FH
70 CALC_OFFSET  EQU 008C0H
71 CARTRIDGE     EQU 08000H
72 CONTROLLER_MAP EQU 08008H
73 DECLSN       EQU 00190H
74 DECHSN       EQU 0019BH
75 DECODER       EQU 01F79H
76 DEFER_WRITES EQU 073C6H
77 EXFOVER      EQU 002EEH
78 ENLARGE      EQU 01F73H
79 ENLRG        EQU 01D6CH
80 FILL_VRAM     EQU 01F82H
81 FREE_SIGNAL  EQU 01FCAH
82 FREE_SIGNALP EQU 01F9DH
83 FREQ_SWEEP   EQU 000FCH
84 GAME_NAME     EQU 0B024H
85 GAME_OPT     EQU 01F7CH
86 GET_BKGRND   EQU 00B9BH
87 GET_VRAM     EQU 01F8BH
88 GET_VRAMP    EQU 01F8EH
89 INIT_SPR_ORDER EQU 01FC1H
90 INIT_SPR_ORDERP EQU 01F94H
91 INIT_TABLE    EQU 01F8BH
92 INIT_TAHLEP  EQU 01F8BH
93 INIT_TIMER    EQU 01FC7H
94 INIT_TIMERP  EQU 01F9AH
95 INIT_WRITER  EQU 01FL5H
96 INIT_WRITERP EQU 01FAFH
97 IRQ_INT_VECT EQU 0801EH
98 LEAVE_EFFECT EQU 001D5H
99 LOAD_ASCII   EQU 01F7FH
100 LOCAL_SPR_TBL EQU 08002H
101 MODE_1       EQU 01F83H
102 MSNTOLSN     EQU 001A6H
103 MUX_SPRITES  EQU 073C7H
104 NMI_INT_VECT EQU 0D021H
105 NUMMR_TABLE EQU 0006CH
106 PLAY_IT      EQU 01FF1H
107 PLAY_ITP     EQU 01F82H
108 PLAY_SONGS   EQU 01FA1H
109 POLLER       EQU 01FLBH
110 PUTFRAME     EQU 00B0BH
111 PUTOBJ       EQU 01FFAH
112 PUTORIP     EQU 01F67H
113 PUT_FRAME    EQU 0B00BH
114 PUT_VRAM     EQU 01F8EH
115 PUT_VRAMP    EQU 01F91H

;CONTROLLE:OS
;PUT_OBJEC:OS
;GAME_OPT:OS
;LOCO:OS
;PUT_MOBIL:OS PUT_SPR:OS
;GAME_OPT:OS LOCO:OS
;GAME_OPT:OS
;TABLE_MA:OS
;GAME_OPT:OS
;TABLE_MA:OS
;PUT_CMPLX:OS
;GAME_OPT:OS LOCO:OS PUT_MOBIL:OS PUT_SPR:OS

```

CONFIDENTIAL

CONFIDENTIAL

LOCATION SUBJECT CODE LINE SOURCE LINE

```

<07E9> 116 PX_TO_PTRN_POS EQU 007E9H
<1FFD> 117 RAND_GEN EQU 01FFDH
<73C8> 118 RAND_NUM EQU 073C8H
<1FDC> 119 READ_REGISTER EQU 01FDCB
<1FE2> 120 READ_VRAM EQU 01FE2H
<1FAC> 121 READ_VRAM EQU 01FACH
<1F6D> 122 REFLECT_HORIZONTAL EQU 01F6DH
<1F6A> 123 REFLECT_VERTICAL EQU 01F6AH
<1FCD> 124 REQUEST_SIGNAL EQU 01FCDH
<1FA0> 125 REQUEST_SIGNALP EQU 01FA0H
<1F70> 126 ROTATE_90 EQU 01F70H
<800F> 127 RST_10H_RAM EQU 0800FH
<8012> 128 RST_18H_RAM EQU 08012H
<8015> 129 RST_20H_RAM EQU 08015H
<8018> 130 RST_28H_RAM EQU 08018H
<801B> 131 RST_30H_RAM EQU 0801BH
<800C> 132 RST_8H_RAM EQU 0800CH
<1FEE> 133 SOUND_INIT EQU 01FEEH
<1FB2> 134 SOUND_INITP EQU 01FB2H
<1FF4> 135 SOUND_MAN EQU 01FF4H
<8004> 136 SPRITE_ORDER EQU 08004H
<73B9> 137 STACK EQU 073B9H
<800A> 138 START_GAME EQU 0800AH
<1FD0> 139 TEST_SIGNAL EQU 01FD0H
<1FA3> 140 TEST_SIGNALP EQU 01FA3H
<1FD3> 141 TIME_MGR EQU 01FD3H
<1FD6> 142 TURN_OFF_SOUND EQU 01FD6H
<1FB8> 143 UPDATE_SPINNER EQU 01FB8H
<73C3> 144 VDP_MODE_WORD EQU 073C3H
<73C5> 145 VDP_STATUS_BYTE EQU 073C5H
<8006> 146 WORK_BUFFER EQU 08006H
<1FEB> 147 WRITER EQU 01FEBH
<1FD9> 148 WRITE_REGISTER EQU 01FD9H
<1FA6> 149 WRITE_REGISTERP EQU 01FA6H
<1FDF> 150 WRITE_VRAM EQU 01FDFH
<1FA9> 151 WRITE_VRAMP EQU 01FA9H
<1FC4> 152 WR_SPR_NH_TBL EQU 01FC4H
<1F97> 153 WR_SPR_NH_TBLP EQU 01F97H
154

```

LOGO:OS
PUT_MORIL:OS

TABLE_MA:OS
LOGO:OS

GRAPHICS:OS VD_DRIVER:OS TABLE_MA:OS PUT_MORIL:OS ACT2:OS
GRAPHICS:OS PUT_MORIL:OS PUT_SPR:OS PUTSEMI2:OS ACT2:OS
GAME_OPT:OS LOGO:OS
GAME_OPT:OS LOGO:OS PUT_MORIL:OS

End of defined reference points

```

156
157
158 GLB ACTIVATE
159 GLB ACTIVATER
160 GLB ADD816
161 GLB AMERICA
162 GLB ASCII_TABLE
163 GLB ATN_SWEEP
164 GLB CALC_OFFSET
165 GLB CARTRIDGE
166 GLB CONTROLLER_MAP
167 GLB DECLSN
168 GLB DECM5N
169 GLB DECODER
170 GLB DEFER_WRITES
171 GLB EFXOVER
172 GLB ENLARGE
173 GLB ENLRG
174 GLB FILL_VRAM
175 GLB FREE_SIGNAL
176 GLB FREE_SIGNALP
177 GLB FREQ_SWEEP
178 GLB GAME_NAME
179 GLB GAME_OPT
180 GLB GET_RKGRND
181 GLB GET_VRAM
182 GLB GET_VRAMP
183 GLB INIT_SPR_ORDER
184 GLB INIT_SPR_ORDERP
185 GLB INIT_TABLE
186 GLB INIT_TABLEP
187 GLB INIT_TIMER
188 GLB INIT_TIMERP
189 GLB INIT_WRITER
190 GLB INIT_WRITERP
191 GLB IRQ_INT_VECT
192 GLB LEAVE_EFFECT
193 GLB LOAD_ASCII
194 GLB LOCAL_SPR_TBL
195 GLB MSNTOLSN
196 GLB MODE_1
197 GLB MUX_SPRITES
198 GLB NMI_INT_VECT
199 GLB NUMBER_TABLE
200 GLB PLAY_IT
201 GLB PLAY_ITP
202 GLB PLAY_SONGS
203 GLB POLLER
204 GLB PUTFRAME
205 GLB PUTOKJ
206 GLB PUTOBJP
207 GLB PUT_FRAME

```

```

;The following defines each access point
; as Global.

```

CONFIDENTIAL

LOCATION	OBJECT CODE	LINE	SOURCE	LINE
		213	GLB READ_REGISTER	
		214	GLB READ_VRAM	
		215	GLB READ_VRAM	
		216	GLB REFLECT_HORIZONTAL	
		217	GLB REFLECT_VERTICAL	
		218	GLB REQUEST_SIGNAL	
		219	GLB REQUEST_SIGNALP	
		220	GLB ROTATE_90	
		221	GLB RST_10H_RAM	
		222	GLB RST_18H_RAM	
		223	GLB RST_20H_RAM	
		224	GLB RST_28H_RAM	
		225	GLB RST_30H_RAM	
		226	GLB RST_8H_RAM	
		227	GLB SOUND_INIT	
		228	GLB SOUND_INITP	
		229	GLB SOUND_MAN	
		230	GLB SPRITE_ORDER	
		231	GLB STACK	
		232	GLB START_GAME	
		233	GLB TEST_SIGNAL	
		234	GLB TEST_SIGNALP	
		235	GLB TIME_MGR	
		236	GLB TURN_OFF_SOUND	
		237	GLB UPDATE_SPINNER	
		238	GLB VDP_MODE_WORD	
		239	GLB VDP_STATUS_BYTE	
		240	GLB WORK_BUFFER	
		241	GLB WRITER	
		242	GLB WRITE_REGISTER	
		243	GLB WRITE_REGISTERP	
		244	GLB WRITE_VRAM	
		245	GLB WRITE_VRAM	
		246	GLB WR_SPR_MM_TBL	
		247	GLB WR_SPR_MM_TBLP	

Errors: 0

CONFIDENTIAL

CONFIDENTIAL

APPENDIX E
 JUMP TABLE

PLAY_SONGS	1F61
ACTIVATEP	1F64
PUTOBJP	1F67
REFLECT_VERTICAL	1F6A
REFLECT_HORIZONTAL	1F6D
ROTATE_90	1F70
ENLARGE	1F73
CONTROLLER_SCAN	1F76
DECODER	1F79
GAME_OPT	1F7C
LOAD_ASCII	1F7F
FILL_VRAM	1F82
MODE_1	1F85
UPDATE_SPINNER	1F88
INIT_TABLEP	1F8B
GET_VRAM	1F8E
PUT_VRAM	1F91
INIT_SPR_ORDERP	1F94
WR_SPR_NH_TBLP	1F97
INIT_TIMERP	1F9A
FREE_SIGNALP	1F9D
REQUEST_SIGNALP	1FA0
TEST_SIGNALP	1FA3
WRITE_REGISTERP	1FA6
WRITE_VRAM	1FA9
READ_VRAMP	1FAC
INIT_WRITERP	1FAF
SOUND_INITP	1FB2
PLAY_ITP	1FB5
INIT_TABLE	1FB8
GET_VRAM	1FBB
PUT_VRAM	1FBE
INIT_SPR_ORDER	1FC1
WR_SPR_NH_TBL	1FC4
INIT_TIMER	1FC7
FREE_SIGNAL	1FCA
REQUEST_SIGNAL	1FCD
TEST_SIGNAL	1FD0
TIME_MGR	1FD3
TURN_OFF_SOUND	1FD6
WRITE_REGISTER	1FD9
READ_REGISTER	1FDC
WRITE_VRAM	1FDF
READ_VRAM	1FE2
INIT_WRITER	1FE5
WRITER	1FE8
POLLER	1FEB
SOUND_INIT	1FEE
PLAY_IT	1FF1
SOUND_MAN	1FF4
ACTIVATE	1FF7
PUTOBJ	1FFA
RAND_GEN	1FFD

APPENDIX F

OS SYMBOLS

1	ACTIVATE	EQU 01FF7H	;	OS:OS
2	ACTIVATEP	EQU 01F64H	;	OS:OS
3	ADD816	EQU 001B1H	;	OS:OS
4	AMERICA	EQU 00069H	;	OS:OS
5	ASCII_TABLE	EQU 0006AH	;	OS:OS
6	ATN_SWEEP	EQU 0012FH	;	OS:OS
7	CARTRIDGE	EQU 08000H	;	OS:OS
8	CONTROLLER_MAP	EQU 08008H	;	OS:OS
9	CTRL_PORT_PTR	EQU 01D43H		
10	DATA_PORT_PTR	EQU 01D47H		
11	DECLSN	EQU 00190H	;	OS:OS
12	DECM5N	EQU 0019BH	;	OS:OS
13	DECODER	EQU 01F79H	;	OS:OS
14	DEFER_WRITES	EQU 073C6H	;	OS:OS
15	EFXOVER	EQU 002EEH	;	OS:OS
16	ENLARGE	EQU 01F73H	;	OS:OS
17	ENLRG	EQU 01D6CH	;	OS:OS
18	FILL_VRAM	EQU 01FB2H	;	OS:OS
19	FREE_SIGNAL	EQU 01FCAH	;	OS:OS
20	FREE_SIGNALP	EQU 01F9DH	;	OS:OS
21	FREQ_SWEEP	EQU 000FCH	;	OS:OS
22	GAME_NAME	EQU 08024H	;	OS:OS
23	GAME_OPT	EQU 01F7CH	;	OS:OS
24	GET_VRAM	EQU 01FB8H	;	OS:OS
25	GET_VRAM_P	EQU 01FB8H	;	OS:OS
26	INIT_SPR_ORDER	EQU 01FC1H	;	OS:OS
	INIT_SPR_ORDERP	EQU 01F94H	;	OS:OS
	INIT_TABLE	EQU 01FB8H	;	OS:OS
	INIT_TABLEP	EQU 01FB8H	;	OS:OS
	INIT_TIMER	EQU 01FC7H	;	OS:OS
	INIT_TIMERP	EQU 01F9AH	;	OS:OS
	INIT_WRITER	EQU 01FE5H	;	OS:OS
	INIT_WRITERP	EQU 01FAFH	;	OS:OS
	IRQ_INT_VECT	EQU 0801EH	;	OS:OS
	LEAVE_EFFECT	EQU 001D5H	;	OS:OS
	LOAD_ASCII	EQU 01F7FH	;	OS:OS
	LOCAL_SPR_TBL	EQU 08002H	;	OS:OS
	MODE_1	EQU 01FB5H	;	OS:OS
	MSNTOLSN	EQU 001A6H	;	OS:OS
	MUX_SPRITES	EQU 073C7H	;	OS:OS
	NMI_INT_VECT	EQU 08021H	;	OS:OS
	NUMBER_TABLE	EQU 0006CH	;	OS:OS
	PLAY_IT	EQU 01FF1H	;	OS:OS
	PLAY_ITP	EQU 01FB5H	;	OS:OS
	PLAY_SONGS	EQU 01F61H	;	OS:OS

1	POLLER	EQU 01FEBH	;	OS:OS
2	PUTOBJ	EQU 01FFAH	;	OS:OS
	PUTOBJP	EQU 01F67H	;	OS:OS
3	PUT_VRAM	EQU 01FBEH	;	OS:OS
	PUT_VRAM_P	EQU 01F91H	;	OS:OS
4	RAND_GEN	EQU 01FFDH	;	OS:OS
	RAND_NUM	EQU 073C8H	;	OS:OS
5	READ_REGISTER	EQU 01FDCH	;	OS:OS
	READ_VRAM	EQU 01FE2H	;	OS:OS
6	READ_VRAM_P	EQU 01FACH	;	OS:OS
	REFLECT_HORIZON	EQU 01F6DH	;	OS:OS
7	REFLECT_VERTICAL	EQU 01F6AH	;	OS:OS
	REQUEST_SIGNAL	EQU 01FCDH	;	OS:OS
8	REQUEST_SIGNAL_P	EQU 01FA0H	;	OS:OS
	ROTATE_90	EQU 01F70H	;	OS:OS
9	RST_10H_RAM	EQU 0800FH	;	OS:OS
	RST_18H_RAM	EQU 08012H	;	OS:OS
10	RST_20H_RAM	EQU 08015H	;	OS:OS
	RST_28H_RAM	EQU 08018H	;	OS:OS
11	RST_30H_RAM	EQU 0801BH	;	OS:OS
	RST_8H_RAM	EQU 0800CH	;	OS:OS
12	SOUND_INIT	EQU 01FEEH	;	OS:OS
	SOUND_INIT_P	EQU 01FB2H	;	OS:OS
13	SOUND_MAN	EQU 01FF4H	;	OS:OS
	SPRITE_ORDER	EQU 08004H	;	OS:OS
14	STACK	EQU 073B9H	;	OS:OS
	START_GAME	EQU 0800AH	;	OS:OS
15	TEST_SIGNAL	EQU 01FD0H	;	OS:OS
	TEST_SIGNAL_P	EQU 01FA3H	;	OS:OS
16	TIME_MGR	EQU 01FD3H	;	OS:OS
	TURN_OFF_SOUND	EQU 01FD6H	;	OS:OS
17	UPDATE_SPINNER	EQU 01F8BH	;	OS:OS
	VDP_MODE_WORD	EQU 073C3H	;	OS:OS
18	VDP_STATUS_BYTE	EQU 073C5H	;	OS:OS
	WORK_BUFFER	EQU 08006H	;	OS:OS
19	WRITER	EQU 01FEBH	;	OS:OS
	WRITE_REGISTER	EQU 01FD9H	;	OS:OS
20	WRITE_REGISTER_P	EQU 01FA6H	;	OS:OS
	WRITE_VRAM	EQU 01FDFH	;	OS:OS
21	WRITE_VRAM_P	EQU 01FA9H	;	OS:OS
	WR_SPR_NH_TBL	EQU 01FC4H	;	OS:OS
22	WR_SPR_NH_TBL_P	EQU 01F97H	;	OS:OS
23				
24				
25				
26				

1	GLB ACTIVATE	;	OS:OS
2	GLB ACTIVATEP	;	OS:OS
3	GLB ADD816	;	OS:OS
4	GLB AMERICA	;	OS:OS
5	GLB ASCII_TABLE	;	OS:OS
6	GLB ATN_SWEEP	;	OS:OS
7	GLB CARTRIDGE	;	OS:OS
8	GLB CONTROLLER_MAP	;	OS:OS
9	GLB CTRL_PORT_PTR	;	OS:OS
10	GLB DATA_PORT_PTR		
11	GLB DECLSN	;	OS:OS
12	GLB DECM SN	;	OS:OS
13	GLB DECODER	;	OS:OS
14	GLB DEFER_WRITES	;	OS:OS
15	GLB EFXOVER	;	OS:OS
16	GLB ENLARGE	;	OS:OS
17	GLB ENLRG	;	OS:OS
18	GLB FILL_VRAM	;	OS:OS
19	GLB FREE_SIGNAL	;	OS:OS
20	GLB FREE_SIGNALP	;	OS:OS
21	GLB FREQ_SWEEP	;	OS:OS
22	GLB GAME_NAME	;	OS:OS
23	GLB GAME_OPT	;	OS:OS
24	GLB GET_VRAM	;	OS:OS
25	GLB GET_VRAMP	;	OS:OS
26	GLB INIT_SPR_ORDER	;	OS:OS
	GLB INIT_SPR_ORDERP	;	OS:OS
	GLB INIT_TABLE	;	OS:OS
	GLB INIT_TABLEP	;	OS:OS
	GLB INIT_TIMER	;	OS:OS
	GLB INIT_TIMERP	;	OS:OS
	GLB INIT_WRITER	;	OS:OS
	GLB INIT_WRITERP	;	OS:OS
	GLB IRQ_INT_VECT	;	OS:OS
	GLB LEAVE_EFFECT	;	OS:OS
	GLB LOAD_ASCII	;	OS:OS
	GLB LOCAL_SPR_TBL	;	OS:OS
	GLB MSNTOLSN	;	OS:OS
	GLB MODE_1	;	OS:OS
	GLB MUX_SPRITES	;	OS:OS
	GLB NMI_INT_VECT	;	OS:OS
	GLB NUMBER_TABLE	;	OS:OS
	GLB PLAY_IT	;	OS:OS
	GLB PLAY_ITP	;	OS:OS
	GLB PLAY_SONGS	;	OS:OS
	GLB POLLER	;	OS:OS
	GLB PUTOBJ	;	OS:OS
	GLB PUTOBJP	;	OS:OS

1		
2	GLB PUT_VRAM	OS:OS
3	GLB PUT_VRAM	OS:OS
4	GLB RAND_GEN	OS:OS
5	GLB RAND_NUM	OS:OS
6	GLB READ_REGISTER	OS:OS
7	GLB READ_VRAM	OS:OS
8	GLB READ_VRAM	OS:OS
9	GLB READ_VRAM	OS:OS
10	GLB REFLECT_HORIZON	OS:OS
11	GLB REFLECT_VERTICAL	OS:OS
12	GLB REQUEST_SIGNAL	OS:OS
13	GLB REQUEST_SIGNALP	OS:OS
14	GLB ROTATE_90	OS:OS
15	GLB RST_10H_RAM	OS:OS
16	GLB RST_18H_RAM	OS:OS
17	GLB RST_20H_RAM	OS:OS
18	GLB RST_28H_RAM	OS:OS
19	GLB RST_30H_RAM	OS:OS
20	GLB RST_8H_RAM	OS:OS
21	GLB SOUND_INIT	OS:OS
22	GLB SOUND_INITP	OS:OS
23	GLB SOUND_MAN	OS:OS
24	GLB SPRITE_ORDER	OS:OS
25	GLB STACK	OS:OS
26	GLB START_GAME	OS:OS
	GLB TEST_SIGNAL	OS:OS
	GLB TEST_SIGNALP	OS:OS
	GLB TIME_MGR	OS:OS
	GLB TURN_OFF_SOUND	OS:OS
	GLB UPDATE_SPINNER	OS:OS
	GLB VDP_MODE_WORD	OS:OS
	GLB VDP_STATUS_BYTE	OS:OS
	GLB WORK_BUFFER	OS:OS
	GLB WRITER	OS:OS
	GLB WRITE_REGISTER	OS:OS
	GLB WRITE_REGISTERP	OS:OS
	GLB WRITE_VRAM	OS:OS
	GLB WRITE_VRAM	OS:OS
	GLB WR_SPR_NM_TBL	OS:OS
	GLB WR_SPR_NM_TBLP	OS:OS

APPENDIX G

TIMING SOFTWARE DATA STRUCTURE

Table Name:

TIMER_TABLE

Description:

A variable length table located in CRAM which consists of an array of three byte entries. Each entry represents a time request.

Access Method:

Pointed to by TIME_TABLE_BASE.

Format:

Each entry appears as:

7	6	5	4	3	2	1	0
D	R	F	E	L	U	U	U
a							
a							

Where:

D: Done
R: Repeat
F: Free
E: Last_Timer_In_Table
L: Long
U: Unused
a: Counter Byte or pointer to a four byte block for long-repeating timers

1 Appendix G (continued)

2 Notes:

3 Done Bit: This bit is set when the counter has
4 finished.

5 Repeat Bit: This bit is set to allow TIME_MGR to
6 restart the counter at its original
value.

7 Free Bit: This bit is set to signify that the
timer is not in use.

8 Last_Timer_In_Table Bit: This bit indicates the last initialized
9 timer in the table.

10 Long Bit: This bit defines the timer type.

11 0 - Short timer
12 1 - Long timer
13
14
15
16
17
18
19
20
21
22
23
24
25
26

APPENDIX H OS SUBROUTINE LIBRARY

NAME	INPUT REGISTERS																DESCRIPTION	SECTION
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P		
ACTIVATE	x	x															Moves pattern and color generator to VRAM	3.3.2.4
ADD16	x	x															Adds 8-bit signed (A) to 16-bit unsigned (HL) → (HL)	9.1
COPT_SCAN																	Moves controller port data to CRAM	6.4
DECIHL	x	x															Decrements least significant nibble pointed to by (HL)	9.2
DECIHL	x	x															Decrements most significant nibble pointed to by (HL)	6.3
DECODER	x	x															Calls COPT_SCAN	6.3
ENLARGE	x	x	x	x	x	x	x										Doubles the size of the original object	3.2.2.4
FILL_VRAM	x	x	x	x													Writes a specific value to VRAM	3.1.5
FREE_SIGNAL																	Releases a timer to the free list based on SIGNAL-SUB	5.8
GET_VRAM	x	x	x	x													Moves VRAM table entry to CRAM	3.2.1.2
INIT_SPR_ORDER																	Initializes SPRITE_ORDER data area to zero	3.2.3.1
INIT_TABLE	x	x															Initializes the VDP base address for given table	3.2.2.1
INIT_TIMER	x	x	x	x													Initializes timer data areas	5.4
INIT_WRITER	x	x															Initializes queue size, head and tail addresses and queue head and tail	4.1
LOAD_ASCII																	Writes ASCII generator set to pattern generator table	9.6
MODE_1																	Sets VDP to graphics mode 1 and sprite size 0	3.1.6
RENTOLSH	x	x															(HL) → byte.HSH to (HL) → byte.LSH	9.4
PLAY_FT																	Called to start a sound	7.3
PLAY_SOUND																	Moves frequency and attenuation data to sound chips	7.5
POLLER																	Reads, decodes and debounces all active portions of both controllers	6.2
PUTOBJ																	Changes an object's frame or location on the display	3.3.4
PUT_VRAM	x	x	x	x													Moves data from CRAM to VRAM table	3.2.1.3
RAND_GEN																	16-bit random number generator	9.5
READ_REGISTER																	Reads and returns the contents of the VDP register	3.1.3
READ_VRAM	x	x	x	x	x												Reads from VRAM writes to buffer in CRAM	3.1.1
REFLECT_HORIZONTAL	x	x	x	x	x												Reflection of generators about the horizontal axis	3.2.2.2
REFLECT_VERTICAL	x	x	x	x	x												Reflection of generator about the vertical axis	3.2.2.1
REQUEST_SIGNAL	x	x															Sets up a timer for the caller	5.6
ROTATE_90	x	x	x	x	x	x											90-degree clockwise rotation of generator	3.2.2.3
SOUND_INIT																	Initializes various sound data areas	7.2
SOUND_RUN																	Called every VDP interrupt, manages sound data base	7.4
TEST_SIGNAL																	Tests for a time-out of a timer	5.7
TIMER_RUN																	Maintains all OS software timers	5.3
UPDATE_SPINNER																	Servicing controller spinner switch interrupts	6.5
WRITER																	Performs deferred PUTOBJ operations	4.2
WRITE_REGISTER																	Writes a value to a selected VDP register	3.1.4
WRITE_VRAM	x	x	x	x	x												Moves data from CRAM to VRAM	3.2.3.2
WR_SPR_ATTR_TBL																	Moves local sprite data to VRAM sprite attribute table	3.2.3.2